Informatique MP

Cours

David Rupprecht
Lycée P. de Fermat
https://www.cpge-fermat.fr

Table des matières

1 Rappels de base sur Python **1**
- 1.1 Les types usuels . . . 1
- 1.2 Structures de contrôle . . . 2
- 1.3 Fonctions . . . 4
- 1.4 Listes . . . 6
- 1.5 Représentation interne des variables en Python . . . 9
- 1.6 Modules . . . 10
- 1.7 Fichiers . . . 11
- 1.8 Le module os . . . 12

2 Méthodes conseillées de programmation **13**
- 2.1 Gestion des entrées, sorties et des erreurs . . . 13
- 2.2 Jeu de tests associé à un programme . . . 14

3 Algorithmes, complexité et preuves **15**
- 3.1 Algorithmes de première année . . . 15
- 3.2 Complexité . . . 17
- 3.3 Preuves de programmes . . . 18

4 Piles et récursivité **23**
- 4.1 Listes et tableaux . . . 23
- 4.2 Piles et files . . . 24
- 4.3 Module Collections et deque . . . 25
- 4.4 Récursivité . . . 26

5 Tris **29**
- 5.1 Généralités . . . 29
- 5.2 Les tris simples . . . 29
- 5.3 Les tris évolués . . . 32
- 5.4 Tri fusion . . . 33

6 Graphes **35**
- 6.1 Définitions, vocabulaire . . . 35
- 6.2 Représentation et utilisation des graphes en Python . . . 36
- 6.3 Les algorithmes à connaître . . . 36
- 6.4 Algorithmes sur les graphes pondérés . . . 40

7 Programmation dynamique **45**
- 7.1 Dictionnaires . . . 45
- 7.2 Programmation dynamique . . . 47
- 7.3 D'autres exemples et exercices . . . 54

8 Intelligence artificielle : apprentissage **59**
- 8.1 Algorithme des k-plus proches voisins avec distance euclidienne . . . 59
- 8.2 Algorithme des k-moyennes . . . 61

9 Théorie des jeux **65**
- 9.1 Présentation . . . 65
- 9.2 Modélisation . . . 65
- 9.3 Stratégies, stratégies gagnantes . . . 66
- 9.4 Heuristique et algorithme min-max . . . 68

10 CCINP MP **71**
10.1 CCINP MP 2024 71
10.2 CCINP MP 2023 71
10.3 CCINP MP 2022 72
10.4 CCINP MP 2021 72
10.5 CCINP MP 2020 73
10.6 CCINP MP 2019 73
10.7 CCINP MP 2018 73
10.8 CCINP MP 2017 74
10.9 CCINP MP 2016 75
10.10CCINP MP 2015 76

Chapitre 1 | Rappels de base sur Python

Ces rappels ne sont pas écrits dans l'ordre - ce n'est pas un cours. Certaines parties utilisent des éléments qui sont rappelés dans les paragraphes suivants.

I | Les types usuels

I.1 | Les entiers et flottants

OPÉRATIONS SUR LES ENTIERS ET FLOTTANTS

`+,-,*`	opérations classiques	
`//`	division entière	
`/`	division flottante	convertit si besoin
`%`	modulo (reste de la division euclidienne)	
`abs(x)`	valeur absolue	int, float, complex
`pow(x,y)` ou `x**y`	puissance	avec int, float et complex

I.2 | Les booléens

- Deux valeurs définies : *True* et *False* (correspondent aux entiers 1 et 0)
- opérateurs :
  - → `bool1 and bool2` : évalue d'abord `bool1` et si l'expression est vraie, évalue alors `bool2` (appelé évaluation paresseuse - très utilisé lors des tests afin notamment de controler qu'une variable prend une valeur acceptable)
  - → `bool1 or bool2` : même principe sur l'évaluation, s'arrête dès qu'une condition est vraie
  - → `not bool1`

I.3 | Chaînes de caractères

Définition d'une chaîne de caractères

- Une chaîne de caractères est définie sous la forme `'chaîne'` ou `"chaîne"`.
- Le caractère d'échappement \ a plusieurs utilisations. Il permet d'écrire sur plusieurs lignes une ligne trop grande mais aussi d'accéder à certains caractères dans les chaînes : pour utiliser les " ou ', pour certains caractères spéciaux (les plus courants sont `\n` pour le retour à la ligne, `\t` pour une tabulation) :

```
>>> C="Bonjour !\nLa chaîne est sur plusieurs lignes\nenfin presque"
>>> C
'Bonjour !\nLa chaîne est sur plusieurs lignes\nenfin presque'
>>> print(C)
Bonjour !
La chaîne est sur plusieurs lignes
enfin presque
```

Une dernière méthode pour saisir des chaînes avec des caractères spéciaux : les triples guillemets ou triples apostrophes. On tape alors tout ce qu'on veut, sur plusieurs lignes si l'on souhaite. Cette façon est notamment utilisée pour documenter convenablement une fonction.

Opérations sur les chaînes de caractères

OPÉRATIONS SUR LES CHAÎNES

`s+t`	concaténation des deux chaînes
`s*n`	création d'une chaîne ou s est copiée n fois
`s[i]`	éléments en position i - le premier est le 0 (uniquement en lecture)
`s[i:j]`	tranche entre l'élément i (inclus) et j (exclu)
`s[i:j:k]`	tranche entre l'élément i (inclus) et j (exclu) avec un pas de k
`len(s)`	longueur de s

```
>>> c1 = 'informatique'
>>> c2 = 'mp'
>>> c1+' '+c2
informatique mp
>>> c2*3
'mpmpmp'
```

Remarque : les chaînes de caractères sont des objets immuables (non mutable), c'est-à-dire qu'une fois créé, l'objet ne peut plus être modifié (toute modification crée un nouvel objet). Notamment, il n'est pas possible de remplacer un caractère de la chaîne `c` via une affectation `c[position]=...`

I.4 | Conversions

Il suffit en général d'une commande sous la forme `type(donnees)` :

```
>>> chaîne='123'
>>> int(chaîne)+3
126
>>> float(chaîne)+3
126.0
>>> str(23+12)
'35'
```

II | Structures de contrôle

II.1 | Affectation

On utilise = pour affecter une variable. **L'expression de droite est tout d'abord évaluée, puis son résultat est affecté à la variable** (voir p.9 pour plus de détails)

On peut effectuer une affectation à plusieurs variables en même temps

```
>>> a,b = 2,'test'
>>> a
2
>>> b
'test'
```

On peut également utiliser n'importe quel itérable pour affecter plusieurs variables successivement (par dépaquetage), du moment qu'on a le bon nombre de variables :

```
>>> a,b = [2,3]
>>> l1,l2,l3 = range(3)
>>> e,f = (2,5)
```

C'est en fait ce qu'il se passe lors de `a,b = 2,3` : la partie droite est évaluée, cela retourne un tuple `(2,3)`, puis l'affectation multiple se fait.

Cela permet, par exemple, d'échanger le contenu de deux variables :

```
a,b = b,a
# la partie droite est évaluée, mise dans un tuple et le résultat est
    affecté par dépaquetage dans a et b
```

II.2 | Tests

Syntaxe

```
if test1:
    instructions
    if sous_test1:
        sous_bloc_interne
    elif sous_test2:
        sous_bloc_second_test
    else:
        sous_bloc_sinon
    instructions
```

UTILISATION DE L'*évaluation paresseuse*

sur un test `if (cond1) and (cond2):`, on détermine d'abord le résultat du premier test. S'il est faux, le second n'est pas évalué. C'est notamment utile lorsqu'on doit vérifier que l'indice dans un tableau est acceptable. Si L est un tableau de taille n, pour s'assurer qu'on n'aura pas d'erreur `Index out of range`, on peut utiliser : `if (i<n) and (L[i]...)`

TESTS ET BOOLÉEN

- le résultat d'un test est un booléen, donc une valeur qui peut être directement utilisée. Par exemple, si on veut créer une fonction qui teste si un nombre est strictement supérieur à 100 :

```
# version maladroite de la fonction
def plus_que_100(x):
    if x>100:
        return True
    else:
        return False
# version plus propre (et concise) de la fonction
def plus_que_100(x):
    return (x>100)
```

- de même, si vous utilisez le résultat d'une fonction qui renvoie un booléen dans les conditions d'un test, c'est assez maladroit de rajouter des conditions ==`True` dans le test...

```
# maladroit
if plus_que_100(y)==True:
# mieux
if plus_que_100(y):
```

II.3 | Les boucles

Boucles inconditionnelles

```
for x in iterateur:
```

où `iterateur` est un objet « itérable », c'est-à-dire qu'il dispose d'une méthode qui permet de parcourir ses éléments les uns après les autres. Pour les objets itérables usuels :

- `range(n)` (entier de 0 à $n-1$), `range(a,b)` (entiers de a à $b-1$), `range(a,b,pas)` (entiers $a+k.pas$ strictement inférieurs à b)
- les listes : x va décrire chacun des éléments de la liste,

- les chaînes : x va décrire chaque caractère de la chaîne

MODIFICATION DU COMPTEUR

on peut modifier la valeur de x à l'intérieur de la boucle, en revanche au passage suivant, x prendra systématiquement la valeur suivante de l'itérateur (si x décrit les entiers de 0 à 9 et x en est à la valeur 4, même si on modifie x, au passage suivant il prendra la valeur suivante à savoir 5)

Remarque : la fonction `range(10)` ne crée pas la liste des entiers de 0 à 9 mais seulement un objet itérable qui va permettre d'obtenir ces entiers les uns après les autres. Si on veut créer la liste de ces entiers, on peut forcer la conversion en liste :

```
>>> L = list(range(10)) ; L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Boules conditionnelles (tant que)

```
while test:
    bloc1
```

Tant que la condition `test` est vérifiée, on effectue le contenu de la boucle. Il vaut mieux que les variables qui interviennent dans la condition soient modifées dans le corps de la boucle si on ne veut pas se lancer dans une boucle infinie...

Sortie d'une boucle

On dispose de `break` et `return` lorsqu'on veut sortir « brutalement » d'une boucle. Le premier sort simplement de la boucle et continue le programme qui suit, le second (uniquement dans une fonction), quitte définitivement la fonction

```
def fonction(n):
    for i in range(n):
        print(i)
        if i>5:
            break
    print("On est sorti")

def fonction2(n):
    for i in range(n):
        print(i)
        if i>5:
            return
    print("On est sorti")
```

```
>>> fonction(10)
0
1
...
5
6
On est sorti
>>> fonction2(10)
0
1
...
5
6
```

III | Fonctions

III.1 | Définitions

```
def f(x):
    traitement
    return(...)    # pas obligatoire
```

Il n'y a pas de différence entre fonction (prend des arguments et retourne le résultat) et procédure (bloc de programme qui a une utilité particulière sans retourner de valeurs), donc une fonction n'est pas obligée de renvoyer quelque chose. Elle peut cependant renvoyer n'importe quel type d'objet (et même renvoyer des objets de types différents suivant les valeurs de x, ce qui n'est pas forcément conseillé)

On peut définir des fonctions prenant plusieurs paramètres en entrée :

```
>>> def f(x,y):
        return x*y
>>> f(3,4)
12
```

III.2 | Autres situations

Définition d'une fonction avec l'opérateur lambda

On peut avoir besoin d'une fonction simple (par exemple pour la transmettre en argument d'une autre) sans avoir envie de lui donner un nom (et la définir avec `def`). Pour cela, on dispose de l'opérateur `lambda`

```
>>> f = lambda x:x*x
>>> f(2)
  4
>>> (lambda x,y : x+y)(3,4)
7
```

Fonction dans une fonction

On peut avoir besoin localement d'une fonction dans une autre (avec même éventuellement un argument de la fonction extérieure qui peut être utilisée). Par exemple, on veut manipuler une fonction qui dépend d'un paramètre :

```
def essai(n):
    def f(x):
    # utilise la valeur de n transmise en argument de essai
        return 1/(n*n+x*x)
    ...
```

III.3 | Variables locales et globales

```
>>> def f():
...     a=3
...     print(a)
>>> a=1
>>> f()
3
>>>a
1
```

La variable `a` affectée dans la fonction `f` est locale à cette fonction. Elle est même totalement oubliée lorsqu'on quitte la fonction :

En résumé (presque exact), on a

- les arguments passés à une fonction sont créés en tant que variable locale,
- toute variable affectée dans une fonction est locale
- une fonction cherche d'abord dans les variables locales à la fonction (ou à celles d'une fonction englobante) puis dans les variables globales,
- on peut préciser qu'on veut utiliser la version globale d'une variable (et ainsi la modifier globalement) en précisant son statut global dans la fonction (avant de l'utiliser) avec le mot clé `global` (sous la forme `global variable`) - c'est évidemment à utiliser plus que le moins possible (on se retrouve avec une variable modifiée sans forcément le savoir... dangereux).

III.4 | Typage des arguments, du retour - Signature d'une fonction

On peut faire apparaître un « typage » des fonctions (c'est notamment le cas dans les sujets de Centrale). Par exemple

```
def multiplie(chaine: str, nombre: int) -> str:
    return nombre*chaine
```

Dans notre utilisation, ce typage n'est qu'une indication (Python de ne va pas contrôler le type des arguments - ça peut être utilisé par d'autres programmes d'analyse ou de vérification de code).

Les types usuelles : `int`, `float`, `str`, `list` (et `list[int]` pour des listes d'entiers), `tuple`, `dict`, `Callable` (pour les fonctions)

Exercice 1.1 ***(renverser une chaine)***

Écrire une fonction qui prend une chaîne en argument et qui renvoie la chaîne renversée. Par exemple `inverse('bonjour')` renvoie `'ruojnob'`.

Exercice 1.2 ***(Palindrome)***

Écrire deux fonctions qui testent si une chaîne est un palindrome (elle se lit dans les deux sens) - l'une pourra utiliser la fonction de l'exercice précédent, l'autre non.

IV | Listes

IV.1 | Création

Une liste est un ensemble ordonné (chaque élément à un numéro d'ordre) d'éléments de types hétérogènes. Elles sont définies entre crochets et les éléments sont séparés par une virgule. On peut créer une liste

- directement : `L = [1,2,3]`
- par conversion d'un objet itérable : `L = list(range(10))`, `L=list("une chaine de caractères")`
- par concaténation de listes : `L = L1+L2`
- par copie d'une liste `L = L1.copy()`
- par concaténation multiple `L = [0]*5` (donne `[0,0,0,0,0]` : très utile pour initialiser une liste à un certain nombre d'éléments).
- par compréhension : `[ fonction(x) for x in iterable ]`
- par compréhension et filtrage : `[ fonction(x) for x in iterable if conditions]`

Par exemple, on veut construire tous les carrés des éléments plus grands que 2 d'une liste

```
>>> l=[0,4,1,-3,7,8]
>>> [x*x for x in l if x>=2]
[16, 49, 64]
```

IV.2 | Accès aux éléments

La numérotation des éléments d'une liste est exactement la même que pour les chaînes, à la différence que cette fois on peut accéder aux élémets à la fois en lecture et en écriture : si `l` est une liste

ACCÈS AUX ÉLÉMENTS

`l[i]`	éléments en position i - le premier est le 0
`l[i:j]`	tranche entre l'élément i (inclus) et j (exclu)
`l[i:j:k]`	tranche entre l'élément i (inclus) et j (exclu) avec un pas de k
`l[i:]`	tranche entre l'élément i (inclus) et la fin
`l[:j]`	tranche entre le début et l'élément j (exclu)
`len(s)`	taille de la liste

IV.3 | Ajout d'un élément et concaténation

- Lorsqu'on veut ajouter en fin d'une liste `l`, on peut le faire grâce à l'opérateur + :

```
>>> l=[4,5,6]
>>> l+[8]
[4, 5, 6, 8]
>>> l
[4, 5, 6]
```

l'opération `l+[8]` crée une nouvelle liste et ne modifie pas `l`. Si on n'affecte pas le résultat, on a perdu la modification. On peut le réaliser entre deux listes, et même concaténer plusieurs fois une même liste :

```
>>> l=[4,5,6]
>>> m=[0,1]
>>> l+m
[4,5,6,0,1]
>>> m*3
[0,1,0,1,0,1]
>>> [0]*10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- Avec la méthode `append` de la classe `list`. Cette fois on modifie effectivement le contenu de la liste

```
>>> l=[4,5,6]
>>> l.append(8)
>>> l
[4, 5, 6, 8]
```

IMPORTANT - CONCATÉNATION ET CRÉATION

- si on connaît la taille à l'avance, autant créer directement la liste de cette taille plutôt que d'ajouter des éléments les uns après les autres (si la liste est de taille N, on la crée avec `L=[0]*N` par exemple)
- pour ajouter un élément à une liste, on utilise la méthode `append` plutôt que la création complète d'une nouvelle liste avec l'opérateur + (sauf raisons valables)

IV.4 | Les n-uplets (tuple)

Il existe un type proche de celui des listes : les tuples (ou n-uplets). On remplace les crochets par des parenthèses lors de leur définition. La grosse différence est que les données ne sont pas modifiables. C'est notamment ce que renvoie un `return` si on donne plusieurs valeurs séparées par des virgules. Les opérations sont similaires à celles sur les listes

```
>>> t = (3,6,9)
>>> s = 2,3  # transformé en tuple
>>> s
(2,3)
>>> t[1]
6
>>> s+t
(2,3,3,6,9)
```

L'avantage d'être non mutable est qu'un tel objet peut être utilisé comme clé d'un dictionnaire.

IV.5 | Listes et fonctions

Pour les explications, lire le paragraphe suivant. Il faut continuer à dissocier la liste et le contenu de la liste dans la fonction appelée. L'exécution de

```
def f(liste):
    liste[1]="modif"
    # la fonction ne retourne rien
```

```
l=[0,1,2]
f(l)
print(l)
```

donne à l'affichage

```
[0, 'modif', 2]
```

- La liste `l` est transmise en argument de la fonction `f` sous un identifiant `liste` local à la fonction est créé et pointe au même endroit que la liste `l`.
- L'affectation `liste[1]="modif"` change le contenu de la liste, mais ne modifie pas son identifiant si bien que `liste` pointe toujours au même endroit que `l`

```
def f(liste):
    print("avant", liste)
    liste=[4,5,6]
    print("après", liste)
```

```
>>> l=[0,1,2]
>>> f(l)
avant [0, 1, 2]
après [4, 5, 6]
>>> l
[0, 1, 2]
```

IV.6 | Quelques méthodes en complément

MÉTHODES SUR LES LISTES

utilisables	
`l.append(a)`	ajoute l'élément `a` à la fin de la liste `l`
`l.pop(indice)`	retourne l'élément d'indice `indice` et le retire de la liste `l`. Par défaut, c'est le dernier élément qui est retiré. Si la liste est vide, la méthode renvoie une erreur
`l.copy()`	copie la liste `l` de façon superficielle (les éléments ne sont pas récursivement copiés)
`l.sort()`	trie la liste `l`
à éviter	
`l.insert(indice,a)`	insert l'élément `a` à l'indice `indice`. Si cet indice dépasse la taille de la liste, l'élément est ajouté à la fin
`l.extend(iter)`	ajoute à la fin de `l` la liste créée à partir de l'objet itérable `iter`
`l.index(a)`	retourne l'indice de l'élément `a` dans la liste `l` s'il est présent dans cette liste et une erreur sinon
`l.count(a)`	compte le nombre d'occurence de l'élément `a` dans la liste `l`

IV.7 | Exercices

Exercice 1.3 ***(Séparation des termes d'une liste d'entiers)***

Écrire une fonction qui sépare une liste d'entiers en deux nouvelles listes. L'une contiendra seulement les nombres pairs de la liste initiale, et l'autre les nombres impairs.

Exercice 1.4 ***(médiane)***

Le but de cet exercice est d'écrire une fonction qui détermine une médiane d'une liste d'entiers (un nombre - pas forcément dans la liste - tel qu'il y ait autant de termes supérieurs et inférieurs à cette valeur)

1. Programmer l'algorithme suivant : on recherche le minimum et le maximum du tableau, on les supprime et on recommence jusqu'à obtenir un tableau de longueur inférieure ou égale à 2. On déduit alors facilement la valeur de la médiane du ou des entiers restants.
2. Quelle est la complexité de cet algorithme ?

Exercice 1.5 ***(listes et arithmétique)***

1. Écrire une fonction qui retourne l'ensemble des diviseurs stricts d'un entier n (différents de 1 et n)
2. Écrire une fonction qui renvoie la somme des carrés des diviseurs stricts de n (et 0 si n est premier).
3. Créer la liste des entiers inférieurs à n qui sont égaux à la somme des carrés de leurs diviseurs stricts. Quelle conjecture peut-on faire ? (et si vous le voulez... montrez le).
4. On appelle nombre parfait un entier qui est égal à la somme de ses diviseurs différents de lui-même. Déterminer tous les nombres parfaits inférieurs à un certain entier n.

Exercice 1.6

Un tableau contient des entiers entre 0 et 500. Déterminer s'ils sont tous distincts. Réfléchir au même problème mais sans savoir quel est l'entier maximal possible.

Exercice 1.7

On se donne une chaîne de caractères `chaine` et un caractère particulier `cara`. On cherche à séparer la chaîne à chaque caractère particulier rencontré et renvoyer les sous-chaînes sous forme de liste.

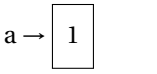
```
>>> separation('Vivement les prochaines vacances',' ')
['Vivement','les','prochaines','vacances']
>>> separation('1.2.3.4.5.6','.')
['1','2','3','4','5','6']
```

V | Représentation interne des variables en Python

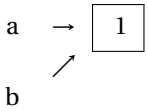
Une partie un peu plus difficile et plus spécifique au langage Python...

```
>>> a=1
>>> b=a
>>> a=2
>>> b
1
```

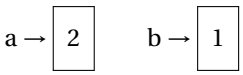
Cela semble tout à fait logique. Que se passe-t-il lorsqu'on valide `a=1` : un emplacement mémoire avec l'entier 1 est créé et a est un alias qui va pointer vers cet emplacement. On peut le représenter ainsi :



Après `b=a`, on se trouve dans la situation

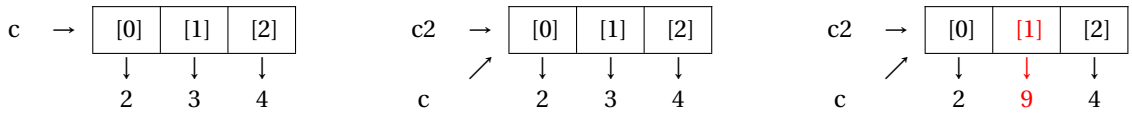


l'affectation finale `a=2` nous amène à la situation



```
>>> c=[2,3,4]
>>> c2=c
>>> c2[1]=9
>>> c
[2, 9, 4]
```

L'évolution des listes c et c2 est représentée ainsi :



Les deux listes c et c2 ont toujours le même identifiant car on n'a pas modifié la liste c2 mais seulement le lien vers la valeur sur laquelle point le deuxième élément de la liste.

VI | Modules

On peut importer des objets d'un module (un fichier d'instructions Python) de différentes façons et à différents endroits de la mémoire :

```
>>> import math
```

Le module `math` est chargé en mémoire et tout est placé dans « un espace de noms », c'est-à-dire (en simplifié) une zone de mémoire propre qui ne se mélange pas avec les données et variables déjà affectées. Pour accéder à un objet de cet espace de noms, on utilsie son préfixe, ici le nom du module chargé :

```
>>> math.sin(0)
0
```

On peut l'importe sous un autre nom

```
>>> import math as m
>>> m.sin(0)
0
```

On peut importer un sous-module d'un module. On le fait fréquemment lorsqu'on utilise de très grosses bibliothèques comme Numpy/Matplotlib

```
>>> import matplotlib.pyplot as plt
```

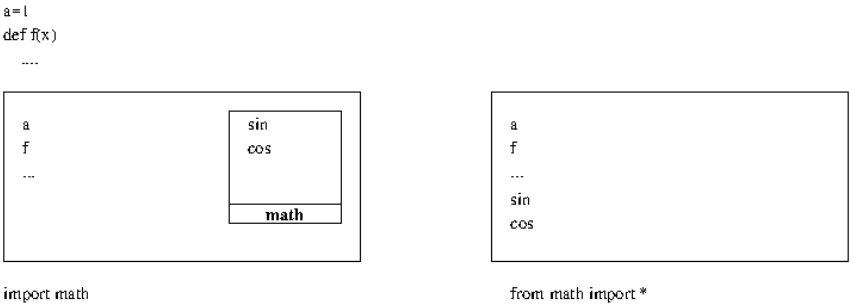
le sous-module `pyplot` de `matplotlib` est importé dans l'espace de noms « plt ».

Enfin, on peut directement importer des fonctions ou des modules complets dans l'espace de noms courant

```
>>> from math import *
# ou
# >>> from math import sin,cos,pi
# pour importer seulement ces fonctions
```

On a alors directement accès aux objets :

```
>>> sin(pi)
1.2246467991473532e-16
```



VI.1 | Module math

- les constantes : `pi, e`
- quelques fonctions classiques : `sqrt, exp, log, log10`, ainsi que `ceil, floor, fabs, factorial` (pour un entier)
- les fonctions trigonométriques diverses : `sin, cos, t an, asin, acos, atan`
- les fonctions hyperboliques : `sinh, cosh, tanh, asinh, acosh, atanh`
- quelques fonctions spéciales : `gamma, lgamma` ($\ln\circ\Gamma$), `erf` (avec $\text{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$)
- quelques autres plus ou moins utilisées...

Il existe un module assez proche pour traiter le cas des fonctions d'une variable complexe (module `cmath`).

VI.2 | Autres modules usuels

MODULES

numpy	pour le calcul numérique essentiellement basé sur le type tableau `array`
matplotlib	tout ce qui concerne l'affichage de graphiques avec plusieurs sous-modules (notamment `matplotlib.pyplot`)
deepcopy	permet la copie « profonde » de listes...
os	pour l'interaction avec le système
random	pour l'aléatoire (on utilisera plutôt `numpy.random`)
time	pour mesurer le temps (obtenir l'heure, le temps d'exécution d'une fonction)

VII | Fichiers

On utilise un fichier en l'ouvrant à partir de la fonction `open`. Cela crée alors un objet avec différentes méthodes. Pour avoir l'aide complète : `help(file)`. Il y a plusieurs paramètres possibles. Les plus standards sont les deux premiers : le nom du fichier et le type d'ouverture (en lecture, en écriture, en ajout...). Par défaut le type est « lecture » (`'r'`). Les autres types usuels sont `'w'` (write) pour l'ouverture en écriture (en vidant le fichier) et `'a'` (append) pour l'ajout à la fin du fichier (et création si le fichier n'existe pas).

Dans cette partie, on va uniquement s'intéresser aux fichiers texte.

VII.1 | Lecture et écriture

Une fois un objet fichier créé, on peut lire les données ou en écrire. À la fin on ferme le fichier :

```
>>> fichier=open("test.txt","w")
>>> fichier.write("Une chaîne dans le fichier\nUne deuxième ligne.")
46   # la méthode retourne le nombre de caractères écrits
>>> fichier.close()
```

Lorsqu'on regarde le contenu du fichier `test.txt`, on obtient (le caractère `\n` est un retour à la ligne).

```
Une chaîne dans le fichier
Une deuxième ligne.
```

On peut ensuite relire ce fichier :

```
>>> fichier=open("test.txt")
>>> a=fichier.read() # on lit l'intégralité du fichier
>>> type(a)
<class 'str'>
>>> a
'Une chaîne dans le fichier\nUne deuxième ligne.'
```

```
>>> print(a)
Une chaîne dans le fichier
Une deuxième ligne.
```

méthode	description
read(nombre)	lit les caractères du fichier, avec un nombre maximal si l'argument est donné. Chaîne vide lorsqu'il n'y a plus rien. Lit l'intégralité si nombre n'est pas précisé
readline()	lit une ligne complète du fichier (retourne une chaine). Vide si à la fin
readlines()	lit les lignes du fichier dans une liste (de chaînes)
write(chaine)	écrit la chaîne dans le fichier
writelines(lignes)	écrit les lignes de la liste transmise (pas de retour à la ligne ajouté à la fin de chaque ligne)
close()	ferme le flux
tell()	indique la position courante dans le fichier

Un fichier texte ouvert en lecture possède également les propriétés d'un itérateur : on parcourt alors les lignes du fichier :

```
fichier=open("test.txt")
for ligne in fichier:   # on parcourt les lignes
    traitement...
```

VIII | Le module os

Voici quelques unes des fonctions du module os qui permettent d'interagir avec le système de fichiers :

fonction	description
getcwd()	répertoire actuel (get current working directory)
chdir(rep)	change le répertoire courant (change directory)
mkdir(rep)	crée un répertoire (make directory)
listdir(chemin)	retourne une liste des fichiers dans le répertoire précisé (répertoire courant par défaut)
remove(chemin)	efface le fichier donné
rename(source,dest)	renomme un fichier ou dossier

Chapitre 2 | Méthodes conseillées de programmation

I | Gestion des entrées, sorties et des erreurs

I.1 | Annotations des fonctions

La définition d'une fonction peut être immédiatement suivie d'une chaîne de caractères (appelée `docstring`) qui permet de documenter la fonction, de préciser les conditions d'entrée et de sortie de la fonction... cette chaîne est utilisée par la commande `help(fonction)`

```
def renverse(t,k):
    """
    Renvoie une nouvelle liste obtenue en inversant les k premiers éléments
       d'une liste
    entrée :
    --------
    t : liste
    k : entier
    sortie :
    --------
    une liste

    Exemple : renverse([1,2,3,4,5,6],3) renvoie  [3,2,1,4,5,6]
    """
```

On prend l'habitude de préciser

- le but de la fonction, avec éventuellement les limitations
- les données en entrée (type, valeurs) et les données en sortie
- éventuellement des exemples significatifs

I.2 | Assertions

Le problème est que cela ne donne que des indications sur les arguments de la fonction. Si elle est appelée avec des arguments d'un mauvais type, elle peut aussi bien provoquer une erreur (ce qui est le mieux qui puisse arriver) ou, pire, continuer en renvoyant un résultat incohérent (parce que les opérations internes à la fonction ne provoquent pas d'erreurs).

Assertions

L'un des moyens simples pour pallier ce défaut de contrôle est d'utiliser la fonction `assert`. Un exemple (plus ou moins pertinent sur les choix des assertions) pour illustrer (avec l'entête de la partie précédente à ajouter) :

```
def renverse(t,k):
    assert(type(t)==list), 'le premier argument doit être une liste'
    assert(len(t)>0), 'la liste doit être non vide'
    assert(k>=2), 'on doit inverser au moins 2 éléments'

    t2 = []                  # nouveau tableau
    n = min(k,len(t))        # nombre d'éléments à vraiment inverser (t trop petit)
    for i in range(n):
        t2.append(t[n-1-i])
    for i in range(n,len(t)):
        t2.append(t[i])
    return t2
```

On teste :

```
>>> renverse([],3)
AssertionError: le tableau doit être non vide
>>> renverse("abcdef",4)
AssertionError: le premier argument doit être une liste
```

II | Jeu de tests associé à un programme

Lorsqu'on écrit un programme, on s'attend à ce qu'il soit correct et renvoie systématiquement la bonne réponse. Ce n'est pas toujours simple de le garantir (voir le chapitre sur les preuves de programmes). On peut se contenter d'avoir un programme qui renvoie la bonne réponse sur certains exemples bien choisis, notamment en s'intéressant aux cas extrêmes.

Considérons l'exemple simple d'un programme qui recherche le maximum d'une liste d'éléments. Quelles sont les situations à envisager ?

- regarder son comportement sur une liste « normale »,
- regarder les cas extrêmes : fonctionne-t-il si le maximum est à l'une des extrémités (début ou fin), s'il y a plusieurs indices pour le maximum ? si la liste n'a qu'un élément ?

Mais on peut aussi envisager d'autres problèmes :

- que se passe-t-il sur une liste vide ?
- que se passe-t-il si on transmet une liste d'éléments qui ne sont pas comparables ?

Ces derniers problèmes sont plutôt d'une autre nature : pour le cas d'une liste vide, il faut faire un choix de valeur de retour dans ce cas (par exemple `None`) ou renvoyer une erreur. Pour le cas d'éléments non comparables, l'erreur viendra toute seule (ou se débrouiller autrement)

Pour valider un programme, on peut donc utiliser un système de tests sur ces différentes situations. Par exemple, vérifier qu'une fonction `maximum` semble correcte :

```
assert(maximum([])==None)
assert(maximum([3])==3)
assert(maximum([3,8,2,4,0,1,4])==8)
assert(maximum([9,4,5,2])==9)
assert(maximum([4,9,2,5,11])==11)
assert(maximum([3,5,8,2,8,1])==8)
```

Si l'une des assertions n'est pas vérifiée, on recevra un message d'erreur. Par exemple la fonction suivante passe les tests

```
def maximum(T):
    """
    renvoie la valeur maximale des éléments de T - on suppose que T possède
       des données comparables

    entrée : liste T
    sortie : la valeur maximale ou None si T est vide
    """
    n = len(T)
    if n==0: return None
    max_temp = T[0]
    for i in range(1,n):
        max_temp = max(max_temp,T[i])
    return max_temp
```

Plus généralement, il y a deux idées principales lorsqu'on veut réaliser un jeu de tests sur un programme :

- **Partionnement** : l'idée est de ne pas tester toutes les entrées possibles mais d'en faire un partitionnement, c'est-à-dire écrire l'ensemble des données d'entrée E en une réunion finie de sous-ensembles disjoints $E = E_1 \cup E_2 \cup \ldots \cup E_k$ et de vérifier que pour chaque sous-ensemble le programme est correcte sur une valeur
- **Tests aux limites** : vérifier que le programme est correcte lorsqu'on le teste avec certaines valeurs limites

Dans l'exemple précédent, on a partitionner l'ensemble des listes en trois sous-ensembles : les listes à 0, 1 et au moins 2 éléments. Les cas limites sont les cas où le maximum est aux extrémités.

Chapitre 3 | Algorithmes, complexité et preuves

I | Algorithmes de première année

Au programme de première année, les algorithmes du programme sont les suivants :

I.1 | Algorithmes simples sur les listes

ou sur les chaînes de caractères (qui est alors vue comme la liste de ses caractères). On fera bien attention de distinguer les boucles sur les éléments et celles sur les positions des éléments (si on parcourt une liste par `for x in L`, la variable `x` va décrire les éléments de `L` les uns après les autres mais on n'a pas accès à la position de l'élément).

- **test d'appartenance à une liste :**

```
def test(L,x):
    for e in L:
        if e==x: return True
        # on quitte la fonction dès qu'on a trouvé l'élément
    # on est sorti de la boucle, on n'a donc pas trouvé
    return False
```

ou

```
def test(L,x):
    for i in range(len(L)):
        if L[i]==x: return True
    return False
```

Dans la première version on parcourt les *éléments* de la liste `L`, alors que dans le second on utilise un compteur entier `i` et on compare avec l'élément en position `i`

- **position d'un élément dans une liste :**

```
def position(L,x):
    for i in range(len(L)):
        if L[i]==x: return i
    return -1
```

- **nombre doccurrences d'un élément :**

```
def nombre(L,x):
    c = 0 # compteur pour compter le nombre de termes
    for e in L:
        if x==e: c+=1
    return c
```

- **recherche du maximum :**

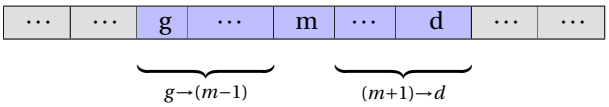
```
def maxi(L):
    M = L[0] # on initialise au premier élément
    for i in range(1,len(L)):
        if L[i]>M: M=L[i] # on met à jour si l'élément est plus grand
    return M
```

- **somme, moyenne des éléments, position du maximum...**

Remarque : on peut parcourir une liste avec les éléments et les positions en même temps : `for i,x in enumerate(L)` décrit la liste `L`, la première variable `i` est un compteur de position et la seconde `x` désigne l'élément actuel. Ce n'est pas au programme, donc à éviter.

I.2 | Recherche dichotomique dans une liste triée

On utilise deux compteurs g et d (gauche/droite) qui donne la portion de la liste à étudier. Pas de grosse difficulté si ce n'est qu'il faut gérer proprement les compteurs pour éviter les boucles infinies (on exclut l'élément milieu s'il n'est pas bon).



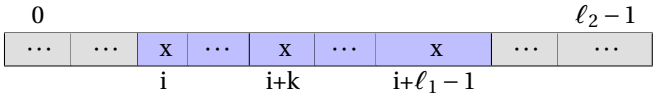

```
def dicho(L,x):
    g = 0
    d = len(L)-1
    # g et d sont initialisés aux extrémités
    while g<=d: # tant qu'il reste des cases à explorer
        m = (g+d)//2
        if L[m]==x: return m
        if L[m]<x: g = m+1 # on explore de m+1 à d
        if L[m]>x: d = m-1 # on explore de g à m-1
    # on est sorti... pas trouvé
    return -1
```

C'est très important de choisir $m+1$ ou $m-1$ sinon lorsque $g=d$, on aurait $m=g=d$ et on bouclerait sans fin si on choisit m au lieu de $m-1$ ou $m+1$.

I.3 | Recherche d'un mot dans une chaîne

On doit pouvoir programmer la recherche basique d'un mot dans une chaîne et compter le nombre d'occurrence d'un mot dans une chaîne (avec ou sans recouvrement possible) en testant position par position si le mot est présent.

- **Recherche d'un mot** : on utilise deux boucles imbriquées, la première (compteur `i`) pour la position du caractère de départ utilisé dans la chaîne et un second compteur (ici `k`) pour décrire l'avancement dans le mot :



Ce n'est pas la peine de chercher si $i+\ell_1-1$ est strictement supérieur à ℓ_2-1 - on fait la recherche tant que $i+\ell_1-1 \leqslant \ell_2-1$, c'est-à-dire $i \leqslant \ell_2-\ell_1$.

```
def recherche(mot,chaîne):
    """
    recherche de la première position d'un mot dans une chaîne
    retourne None si le mot n'apparaît pas
    """
    l1=len(mot)
    l2=len(chaîne)
    i=0
    while i<=l2-l1:
        k=0
        while (k<l1) and (chaîne[i+k]==mot[k]):
        # tant qu'on ne dépasse pas et que les caractères correspondent, on avance
            k+=1
        if k==l1: # sortie car tous les caractères correspondent
            return(i)
        i+=1   # départ suivant
    return(-1)
```

- **Nombre d'occurrences avec chevauchement** : c'est le plus simple : on parcourt les positions de départ les unes après les autres et on incrémente un compteur à chaque fois qu'on voit le mot. Il suffit de modifier la fonction précédente :

```
def occurrence(mot,chaîne):
    l1=len(mot)
    l2=len(chaîne)
    i=0
    c=0
    while i<=l2-l1:
        k=0
        while (k<l1) and (chaîne[i+k]==mot[k]):
        # tant qu'on ne dépasse pas et que les caractères correspondent, on avance
            k+=1
        if k==l1: # sortie car tous les caractères correspondent
            c += 1
        i+=1  # départ suivant
    return(c)
```

On peut remplacer la boucle `while` par une boucle `for` puisqu'on sait qu'on parcourt toutes les positions les unes après les autres, jusqu'au bout.

- **Nombre d'occurrences sans chevauchement** : si on rencontre le mot on doit se placer après :

```
def occurrence(mot,chaîne):
    l1=len(mot)
    l2=len(chaîne)
    i=0
    c=0
    while i<=l2-l1:
        k=0
        while (k<l1) and (chaîne[i+k]==mot[k]):
        # tant qu'on ne dépasse pas et que les caractères correspondent, on avance
            k+=1
        if k==l1: # sortie car tous les caractères correspondent
            c += 1
            i = i+l1 # nouveau départ
        else:
            i+=1
    return(c)
```

II | Complexité

II.1 | Généralités

Beaucoup de questions se posent pour évaluer la complexité d'un algorithme en fonction des données d'entrée :

- **taille des données d'entrée** : que prend-on en compte ? pour une chaîne de caractères, le nombre de caractères semble convenir. Pour un tableau, on pense à son nombre d'éléments mais ce n'est pas toujours suffisant : si c'est un tableau de chaînes de caractères, doit-on prendre le nombre de chaînes ou le nombre total de caractères ? De même pour un tableau d'entiers (et même pour un entier seul). Si les entiers sont codés sur un nombre fixé de bits, leur nombre suffit. Si les entiers deviennent de taille quelconque, leur taille rentre aussi en jeu....
- **type de complexité** : temporelle (le temps que cela prend en fonction de la taille `n` des entrées) et/ou spatiale (la place mémoire totale utilisée par l'algorithme). On ne s'intéressera qu'à la complexité temporelle
- **valeur de la complexité** : minimale, maximale ou moyenne ? Si on note D_n l'ensemble des entrées possibles de taille n et $C(d)$ le coût pour $d \in D_n$ de l'algorithme, on a

$$C_{max} = \max\{C(d), d \in D_n\}, C_{min} = \min\{C(d), d \in D_n\} \text{ et } C_{moy} = \sum_{d \in D_n} C(d)\mu(d),$$

où $\mu(d)$ représente la probabilité d'avoir l'entrée d. On regardera essentiellement la complexité maximale (ce qui peut se passer dans le pire des cas) et, lorsque c'est possible, la complexité moyenne (souvent difficile à calculer mais intéressante lorsqu'on doit utiliser l'algorithme un grand nombre de fois).

II.2 | Exemple et notations

	cout	nombre
`def maxi(L):`	C_0	1
`    n = len(L)`	C_1	1
`    M = L[0]`	C_2	1
`    for i in range(1,n):`	C_3	$n-1$
`        if L[i]>M:`	C_4	$n-1$
`            M = L[i]`	C_5	k
`    return(M)`	C_6	1

Sur cet exemple, le coût total pour une liste à n éléments est $C(n) = C_0 + C_1 + C_2 + C_6 + (C_3 + C_4) * (n-1) + C_5 * k$ où k varie de 0 à $n-1$. On a un coût sous la forme $C(n) = A + kB + nC$. Comme on ne connaît pas les constantes ni la valeur de k, on ne peut donner qu'une complexité asymptotique. La plupart du temps l'ordre de la complexité suffit. On note (tout est positif ici)

- $C(n) = O(f(n))$ lorsqu'il existe une constante K et $n_0 \in \mathbb{N}$ telle que $C(n) \leqslant K.f(n)$ pour $n \geqslant n_0$ (majoration de la complexité asymptotique),
- $C(n) = \Omega(f(n))$ lorsqu'il existe une constante K et $n_0 \in \mathbb{N}$ telle que $C(n) \geqslant K.f(n)$ pour $n \geqslant n_0$ (minoration de la complexité asymptotique),
- $C(n) = \theta(f(n))$ lorsqu'il existe des constantes K_1, K_2 et $n_0 \in \mathbb{N}$ telle que $K_1 f(n) \leqslant C(n) \leqslant K_2 f(n)$ pour $n \geqslant n_0$ (encadrement de la complexité asymptotique),
- on peut évidemment utiliser les équivalents si on veut être précis sur une complexité (par exemple le nombre de comparaisons, le nombre d'affectations...)

Dans l'exemple précédent, on a $C(n) = \theta(n)$. Au programme, on s'intéresse essentiellement à une majoration, d'où $C(n) = O(n)$ ici.

II.3 | Ordres de grandeur

On peut par exemple mesurer les rapidités de calcul en `flops` (floating point operations per second) - un bon PC actuel étant de l'ordre de 10^{11} `flops`. Prenons seulement 10^9 comme exemple

nom	ordre	$n = 10^2$	$n = 10^3$	$n = 10^6$
logarithmique	$\log n$	immédiat	-	-
linéaire	n	-	-	1 ms
quasi-linéaire	$n \log n$	-	-	20 ms
quadratique	n^2	-	-	15 min
cubique	n^3	-	1s	30 ans
exponentielle	a^n	10^{15} ans avec 2^n	/	/

Pour des algorithmes exponentiels (complexité en 2^n par exemple), on peut difficilement dépasser $n = 30$ ou 40... c'est peu.

III | Preuves de programmes

Pour prouver qu'une boucle réalise bien toujours ce qu'on espère :

- on détermine un **invariant de boucle**, c'est-à-dire une propriété qui est vraie à chaque passage de la boucle. En général, cette propriété est la description du contenu des variables importantes lors des différents passages dans la boucle. La propriété peut ou non fait apparaître le nombre de passage dans la boucle.
- on prouve la **terminaison** : on doit sortir de la boucle. C'est immédiat sur une boucle non conditionnelle `for` (nombre d'étapes déterminé) mais parfois plus difficile sur une boucle conditionnelle `while` - la plupart du temps on explicite un variant de boucle : une quantité *entière et strictement décroissante* qui doit rester supérieure à une valeur pour ne pas sortir.
- on prouve la **correction** de la boucle : on vérifie que la sortie correspond à ce qu'on veut

III.1 | Boucle non conditionnelle : calcul de $n!$

Calcul de $n!$

Données : n entier positif ou nul
Sorties : $n!$
début
  $p \leftarrow 1$
  pour i *de* 1 *à* n **faire**
    $p \leftarrow p * i$
  retourner p

Preuve de l'algorithme :

- si $n = 0$, on ne rentre pas dans la boucle et on renvoie $1 = 0!$. On suppose $n \geqslant 1$ pour la suite.
- On note p_k le contenu de p à la fin du k-ième passage, avec $p_0 = 1$. On a l'invariant « $p_k = k!$ ». En effet, c'est vrai pour $k = 0$. Si $p_k = k!$ alors au passage suivant dans la boucle, le compteur i prend la valeur $k+1$ et $p_{k+1} = p_k * (k+1) = (k+1)!$.
- la terminaison est immédiate puisque la boucle est une boucle for avec n étapes.
- correction : on effectue n passages dans la boucle. À la sortie $p = p_n = n!$.

III.2 | Boucle conditionnelle : division euclidienne

Division euclidienne

Données : a entier positif, b entier strictement positif
Sorties : q, r quotient et reste de la division euclidienne de a par b
début
  $q \leftarrow 0$
  $r \leftarrow a$
  tant que $r \geqslant b$ **faire**
    $r \leftarrow r - b$
    $q \leftarrow q + 1$
  retourner *(q,r)*

On note

- q_0 et r_0 la valeur de q et r avant le premier passage dans la boucle
- q_i et r_i la valeur de q et r après le i-ème passage dans la boucle.

Première preuve de l'algorithme :

- invariants : « $q_i = i$, $r_i = r - i * b$ ».
  → la propriété est vraie pour $i = 0$: on a $q_0 = 0$ et $r_0 = a$.
  → la propriété se conserve : si on a effectué i passages dans la boucle et qu'on rentre dans un passage supplémentaire alors $r_i \geqslant b$. On a alors $q_{i+1} = q_i + 1 = i + 1$, ainsi que $r_{i+1} = r_i - b = r - i.b - b = r - (i+1) * b$.
- terminaison : la suite (r_n) est strictement décroissante car $b > 0$ donc il existe i tel que $r_i < b$.
- correction : si n désigne le nombre de passages dans la boucle, alors $q_n = n$, $r_n = a - q_n b$ et $r_n < b$ puisqu'on est sorti de la boucle. On a bien $a = bq_n + r_n$ avec $r_n < b$ mais rien ne donne $r_n \geqslant 0$. Pour cela il faudrait dire qu'on n'était pas sorti avant donc $r_{n-1} \geqslant b$ et ainsi $r_n \geqslant 0$... enfin sauf s'il n'y a pas d'étape $n-1$ car alors... bon on préfère passer à la seconde version en modifiant l'invariant.

Seconde preuve de l'algorithme :

- invariants : « $q_i = i$, $r_i = r - i * b$ et $r_i \geqslant 0$ ».

→ la propriété est vraie pour $i = 0$: on a $q_0 = 0$ et $r_0 = a$ et $r_0 \geqslant 0$.

→ la propriété se conserve : si on a effectué i passages dans la boucle et qu'on rentre dans un passage supplémentaire alors $r_i \geqslant b$. On a alors $q_{i+1} = q_i + 1 = i + 1$, ainsi que $r_{i+1} = r_i - b = r - i.b - b = r - (i+1) * b$ et enfin $r_{i+i} = r_i - b \geqslant b - b = 0$.

- terminaison : la suite (r_n) est strictement décroissante car $b > 0$ donc il existe i tel que $r_i < b$.
- correction : si n désigne le nombre de passages dans la boucle, alors $q_n = n$, $r_n = a - q_n b$, $r_n \geqslant 0$ et $r_n < b$ puisqu'on est sorti de la boucle. On a bien $a = bq_n + r_n$ avec $0 \leqslant r_n < b$.

Troisième preuve de l'algorithme :

- On n'a pas besoin de connaître complètement le contenu de toutes les variables. On propose les invariants : « $a = bq_i + r_i$ et $r_i \geqslant 0$ ».

→ la propriété est vraie pour $i = 0$: on a $q_0 = 0$ et $r_0 = a$ donc $bq_0 + r_0 = a$ et $r_0 \geqslant 0$.

→ la propriété se conserve : si on a effectué i passages dans la boucle et qu'on rentre dans un passage supplémentaire alors $r_i \geqslant b$. On a alors $r_{i+1} = r_i - b$ et $q_{i+1} = q_i + 1$ d'où $bq_{i+1} + r_{i+1} = b(q_i + 1) + (r_i - b) = bq_i + r_i = a$ et $r_{i+1} \geqslant 0$ car $r_i \geqslant b$.

- terminaison : la suite (r_n) est strictement décroissante car $b > 0$ donc il existe i tel que $r_i < b$.
- correction : si n désigne le nombre de passages dans la boucle, alors $bq_n + r_n = a$ et $0 \leqslant r_n < b$ puisqu'on n'est pas rentré dans la boucle après le passage n.

III.3 | Exercices

Exercice 3.8

Suite de Fibonacci On définit la suite (u_n) par $u_0 = 1$, $u_1 = 1$ et, pour tout $n \in \mathbb{N}$, $u_{n+2} = u_{n+1} + u_n$. On propose l'algorithme suivant :

Calcul de u_n

Données : n entier positif ou nul
Sorties : u_n terme d'indice n de la suite de Fibonacci
début
  $a \leftarrow 1$
  $b \leftarrow 1$
  pour i *de* 1 *à* n **faire**
    $a, b \leftarrow b, a + b$
  retourner b

Cet algorithme calcule-t-il ce qu'il est censé calculer dans toutes les situations ? est-il correct ?

Exercice 3.9

Évaluation d'un polynôme

On se donne un polynôme $P = \sum_{k=0}^{n} a_k X^k$ et veut l'évaluer en x. On propose deux algorithmes.

- le premier se décompose en deux fonctions :

`expo(x,k)` : calcul de x^k

Données : x réel, k entier positif ou nul
Sorties : la valeur de x^k
début
  $p \leftarrow 1$
  pour i *de* 1 *à* k **faire**
    $p \leftarrow p * x$
  retourner p

Évaluation d'un polynôme en x

Données : un tableau de coefficient
$a = [a_0, a_1, \ldots, a_n]$ et un réel x
Sorties : la valeur de $\sum_{k=0}^{n} a_k x^k$
début
  $s \leftarrow 0$
  pour i *de* 0 *à* n **faire**
    $s \leftarrow s + a[i] * expo(x, i)$
  retourner s

Justifier que les deux fonctions sont correctes. Déterminer le nombre d'opérations (multiplications et additions) pour cette évaluation. Quel est l'ordre de grandeur ?

- **Algorithme de Hörner** : on remarque que

$$P(x) = ((((a_n \times x + a_{n-1}) \times x) + a_{n-2}) \times x \ldots + a_1) \times x + a_0,$$

on commence par $s = a_n$, puis $s = s.x + a_{n-1}\ldots$. Écrire un algorithme effectuant ce calcul, prouver qu'il est correct et étudier sa complexité.

Exercice 3.10 ***(PGCD)***

Écrire et justifier un algorithme déterminant le pgcd d de 2 entiers p et q, puis un algorithme calculant une relation de Bezout $d = up + vq$.

Chapitre 4 | Piles et récursivité

Il est naturel d'utiliser les listes Python qui servent à peu près à tout... hélas pas mal de difficultés cachées.

I | Listes et tableaux

On distingue essentiellement deux types d'objets :

- les tableaux dont la taille est fixe
- les listes (dynamiques) dont le nombre d'objets est variable

De plus, les éléments peuvent être homogènes (tous de même type) ou hétérogènes (types différents et donc chaque élément a une taille mémoire différente).

I.1 | Tableaux homogènes

C'est la structure la plus simple : le nombre de cases est fixé et chaque case prend la même place. Cela permet de réserver un espace mémoire suffisant et surtout contigu pour stocker les objets. On a alors les possibilités suivantes :

- accès simple et rapide à l'élément en position k : l'adresse est `debut+k*taille`, aussi bien en lecture qu'en écriture
- pas de possibilité d'insertion, d'ajout, de suppression (on peut insérer en décalant les éléments mais il faut alors les déplacer les uns après les autres, éventuellement en supprimant la queue)

les tableaux hétérogènes sont moins intéressants car chaque élément a une taille différente donc il n'y a pas de formule simple pour connaître l'emplacement de l'élément k (si ce n'est d'avancer case par case en fonction de leurs tailles).

I.2 | Listes simplement chaînées

On dispose de l'adresse de début de la liste, chaque élément est séparé en deux parties : la donnée et l'adresse de l'élément suivant.

- on n'a pas de limitation de taille (à part la mémoire), pas de déclaration de taille à faire
- on a facilement accès à l'élément de début
- pour aller à l'élément k, on doit parcourir tous les éléments précédents
- on peut facilement insérer et supprimer un élément en tête de liste
- on peut insérer un élément à une position une fois qu'on est à cette position mais on ne peut pas supprimer au milieu (on n'a pas l'adresse précédente)

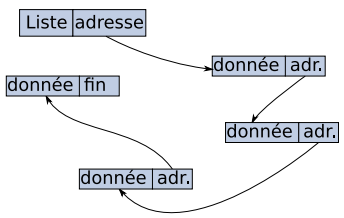


FIGURE 4.1 – Liste chaînée

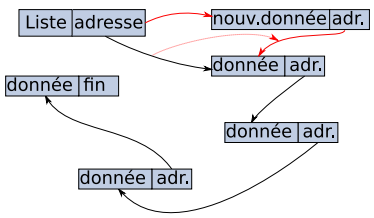


FIGURE 4.2 – Insertion au début

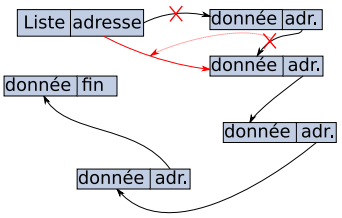


FIGURE 4.3 – Suppression au début

I.3 | Listes doublement chaînées

C'est le même principe mais chaque élément dispose en plus de l'adresse de l'élément précédent (et l'élément précédent le premier élément est le dernier élément : on a alors un cycle).

- on n'a pas de limitation de taille (à part la mémoire), pas de déclaration de taille à faire
- on a facilement accès à l'élément de début et de fin
- pour aller à l'élément k, on doit parcourir tous les éléments précédents
- on peut facilement insérer et supprimer un élément en tête et en queue de liste
- on peut insérer et supprimer un élément à une position une fois qu'on est à cette position

I.4 | En Python

le type `list` en Python est dynamique. L'opération essentielle est d'accéder rapidement aux éléments, ce qui est facile pour les tableaux homogènes. Le choix est le suivant :

- on utilise un tableau d'adresses (ainsi les tailles sont fixes) d'une certaine taille. Un élément est alors une adresse vers l'objet correspondant. Cela permet d'accéder en lecture et écriture à un élément quelconque rapidement
- lorsqu'on veut ajouter un élément en fin de liste : soit il reste des cases vides dans le tableau d'adresses, dans ce cas pas de difficulté ; soit tout le tableau est rempli et dans ce cas on alloue une zone mémoire plus grande, on recopie le tableau dans cette nouvelle zone et on ajoute l'élément (si on ajoute une fois, il y a des chances qu'on ajoute plusieurs fois. Plutôt que d'ajouter une seule case au tableau, on augmente sa taille suffisamment pour des insertions futures - ni trop pour ne pas gaspiller de mémoire, ni trop peu pour ne pas avoir à le refaire souvent). La complexité de l'insertion est alors presque constante (on dit en temps amorti constant).
- Supprimer le dernier élément est facile, supprimer un élément intermédiaire beaucoup plus long car il faut décaler toutes les cases...

Pour concaténer deux listes, on en crée une suffisant longue et on copie les adresses (complexité de l'ordre de la taille totale).

II | Piles et files

On rappelle ici les deux structures de données : piles (LIFO : Last In, First Out) et files (FIFO : First In, First Out) qui correspondent bien à leurs noms... on empile sur une pile, le dernier élément empilé et au dessus et c'est celui auquel on a accès ; pour les files, on peut se la représenter comme une file d'attente, les éléments arrivent les uns après les autres mais c'est le premier arrivé qui est ensuite utilisé.

II.1 | Piles

Opérations sur les piles

C'est une structure de données dont les opérations sont les suivantes :

- créer une pile vide,
- savoir si la pile est vide,
- ajouter un objet en sommet de pile,
- récupérer et supprimer l'objet en haut de la pile

Les exemples usuels : une pile d'assiettes, l'historique d'un navigateur... La structure de données de liste simplement chaînée se prête parfaitement à la réalisation d'une pile.

En Python basique

On commence par une présentation utilisant les listes Python (l'avantage : c'est simplet et intégré à Python, l'inconvénient : ce n'est pas la structure optimale adaptée pour gérer les piles)

- on ajoute un élément avec la méthode `append` : `L.append(x)`
- on dépile le dernier élément avec `pop` : `x = L.pop()` (retire le sommet de la pile et le renvoie),
- on peut tester si la pile est vide : `L == []` (ou `len(L)==0`)

- éventuellement (ça dépend), on peut lire le sommet de la pile sans le dépiler (mais cela peut se faire avec un dépilement/empilement)
- on interdit tout le reste (retirer un élément quelconque, demander la taille de la liste, modifier un élément en position quelconque)

On pourra éventuellement créer des fonctions correspondantes pour des raisons de clarté :

```
def pile_vide():
    return []

def est_vide(L):
    return (len(L)==0)

def empiler(L,x):
    L.append(x)

def depiler(L):
    return L.pop()
```

Pile d'exécution / Pile d'appel

Lors de l'appel d'une fonction en Python, un ensemble de données est stocké sur une pile (appelée pile d'exécution) : arguments de la fonction, adresse de retour une fois la fonction terminée, de la place pour les variables locales, pour la valeur de retour... cela permet de gérer l'appel à une fonction et aussi l'appel multiple de fonctions.

II.2 | Files

Opérations sur les piles

Les opérations sont les suivantes :

- créer une file vide,
- savoir si la file est vide,
- ajouter un objet en fine de file,
- récupérer et supprimer l'objet au début de la pile

Cette fois, la structure de données de liste doublement chaînée se prête à la réalisation d'une file : on a besoin de réaliser des opérations à chaque extrémité.

```
def file_vide():
    return []

def est_vide(F):
    return (len(F)==0)

def emfiler(F,x):
    F.append(x)

def defiler(F):
    return F.pop(0) # ça c'est terrible !!!
```

III | Module Collections et deque

Ce module rajoute plusieurs types de structures qui étendent les types basiques de Python (`dict`, `lict`, `set`, `tuple`). On décrit ici le sous-module `deque` (se prononce dèque - vient de l'abréviation double-ended queue) qui permet de gérer les piles et les files de manière optimisée.

```
>>> import collections
```

On peut, pour simplifier l'utilisation du sous-module, importer simplement le sous-module intéressant :

```
>>> from collections import deque
```

On a alors différents moyens pour construire un objet, par le constructeur de type (en lui fournissant éventuellement un itérable)

```
>>> A = deque()         # constructeur vide
>>> B = deque([3,6,9])
>>> C = deque(range(8))
>>> D = deque([],10)    # le seconde argument est la taille maximale du deque
```

On dispose alors de différentes méthodes d'ajouts/retraits

méthode	description
`append(x)`	ajoute l'élément x à la fin
`appendleft(x)`	ajoute l'élément x au début
`extend(iterable)|`	ajoute les éléments de l'itérable à la fin
`extendleft(iterable)`	ajoute en début les éléments de l'itérable l'un après l'autre (donc ils sont dans l'ordre inverse)
`pop()`	retire et renvoie l'élément de fin
`popleft()`	retire et renvoie l'élément de début

Quelques exemples d'utilisation :

```
>>> C = deque(range(8))
>>> C.append(21) ; C.appendleft(-4) ; C
deque([-4, 0, 1, 2, 3, 4, 5, 6, 7, 21])
>>> C.pop()
21
>>> C
deque([-4, 0, 1, 2, 3, 4, 5, 6, 7])
```

On peut parcourir les éléments d'un `deque` avec une boucle for (c'est aussi un itérable :

```
S = 0
for x in C:
    S += x
S
```

renvoie la somme des termes de `C`.
Enfin quelques méthodes supplémentaires sur un `deque` :

méthode	description
`clear()`	efface le contenu
`copy()`	crée une copie (superficielle) du deque
`count(x)`	compte le nombre d'éléments égaux à `x`
`rotate(n=1)`	décale les éléments du `deque` de `n` (1 par défaut)
`reverse()`	inverse les éléments

On peut également accéder aux éléments comme dans une liste (avec `[position]`), avoir la longueur... la complexité d'accès à un élément est en $O(1)$ aux extrémités (c'est fait pour) mais en $O(n)$ au milieu.

IV | Récursivité

IV.1 | Généralités

Il arrive fréquemment qu'une fonction s'appelle elle-même avec d'autres valeurs des arguments - cela ne pose pas de problème grâce à la pile d'appels. Le principe générale d'une fonction récursive est la suivante :

```
def f(x):
    # traitement des cas terminaux immédiats
    if x==...:
```

```
        return val1
    if x==...:
        return val2
    # autres situations
    # avec des calculs faisant appel à f(.)
    return ...
```

IV.2 | Exemple : calcul de puissances

On est souvent proche de la définition mathématique pour la fonction. Par exemple pour $a \in \mathbb{R}$ et $n \in \mathbb{N}$, on a

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ a * a^{n-1} & \text{si } n \geqslant 1 \end{cases}$$

ce qui donne

```
def puissance(a,n):
    if n==0:
        return 1
    else:
        return a*puissance(a,n-1)
```

La preuve est simple à réaliser, elle se fait par récurrence sur n. On note $\mathscr{P}(n)$: « pour tout a, `puissance(a,n)` renvoie a^n ».

1. Pour $n = 0$, `puissance(a,n)` renvoie 1.
2. Soit $n \in \mathbb{N}^*$ tel que $\mathscr{P}$ est vraie jusqu'au rang $n-1$, alors l'appel à `puissance(a,n)` rentre dans le `else` (puisque $n \in \mathbb{N}^*$), et retourne a*`puissance`$(a, n-1) = a * a^{n-1} = a^n$.
3. Par récurrence, la fonction est correcte.

Évidemment ce n'est pas toujours aussi simple.

Concernant la complexité : si on note $T(n)$ la complexité du calcul de a^n, on a une relation de récurrence $T(n) = T(n-1)+C$ où C est constant (à peu près...), d'ou une suite arithmétique et $T(n) = O(n)$. On peut grandement améliorer cela par différents algorithmes (exponentiation rapide en TD par exemple).

IV.3 | Suite de Fibonacci

Première version

On a $u_0 = u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$ pour $n \in \mathbb{N}$. On en déduit une première version :

```
def fibo(n):
    if (n==0) or (n==1):
        return 1
    else:
        return fibo(n-1)+fibo(n-2)
```

Pour la preuve, cela se fait par récurrence comme précédemment. La complexité est plus grande... on va compter le nombre d'additions qu'on réalise. On note A_n ce nombre d'additions pour calculer u_n. On a $A_0 = A_1 = 0$ puis $A_n = 1 + A_{n-1} + A_{n-2}$, ce qui peut se réécrire $(1 + A_n) = (1 + A_{n-1}) + (1 + A_{n-2})$. En posant $a_n = 1 + A_n$, on a $a_0 = a_1 = 1$ et $a_n = a_{n-1} + a_{n-2}$ d'où $a_n = u_n = \theta(\rho^n)$ où $\rho = \frac{1+\sqrt{5}}{2}$. La complexité est exponentielle!!! de plus le nombre d'additions correspond aussi, à peu près, au nombre d'appels de la fonction... la pile d'appel va exploser dès qu'on veut calculer les termes aux delà du 30ème ou 40ème rang.

Seconde version : non récursive

On peut calculer les termes de proche en proche par couple puisqu'on a besoin de deux termes pour avoir le suivant : $(u_n, u_{n+1}) \to (u_{n+1}, u_{n+2} = u_n + u_{n+1})$:

```
def fibo(n):
    a,b = 1,1
    for i in range(n):
        a,b = b,a+b
    return a
```

La preuve : à la fin du passage k, on a $a_k = u_k$ et $b_k = u_{k+1}$. La complexité est cette fois linéaire.

Troisième version : récursive linéaire

Même principe : si on sait calculer le couple (u_n, u_{n+1}), on sait facilement calculer le suivant. On crée une fonction `fibo(n)` qui renvoie le **couple** (u_n, u_{n+1})

```
def fibo(n):
    if n==0:
        return (1,1)
        # on n'a pas le choix, on doit systématiquement renvoyer un couple
    else:
        a,b = fibo(n-1)
        return b,a+b
```

et on récupère le premier élément de la liste (on prouve par récurrence que `fibo(n)` renvoie le couple (u_n, u_{n+1})). Si on note A_n le nombre d'additions, on a immédiatement $A_n = A_{n-1} + 1$. La complexité est linéaire.

Troisième version bis

On veut quand même récupérer u_n et pas un couple... on peut le faire en deux fonctions

```
def fibo_rec(n):
    if n==0:
        return (1,1)
        # on n'a pas le choix, on doit systématiquement renvoyer un couple
    else:
        a,b = fibo_rec(n-1)
        return b,a+b
def fibo(n):
    u = fibo_rec(n)
    return u[0]
```

mais cela crée une fonction `fibo_rec` dans l'espace de noms usuels ce qui n'est pas satisfaisant. On va donc cacher cette fonction dans la fonction principale.

```
def fibo(n):
    def fibo_rec(n):
        if n==0:
            return (1,1)
        else:
            a,b = fibo_rec(n-1)
            return b,a+b
    u = fibo_rec(n)
    return u[0]
```

Chapitre 5 | Tris

I | Généralités

I.1 | Présentation

On cherche à trier une liste de données (pour une relation d'ordre totale). On peut éventuellement faire la théorie sur des tableaux d'entiers (quitte à associer une clé entière à un objet) et même se limiter à trier un tableau d'entiers de 0 à $n-1$ si on veut encore plus simplifier). Plusieurs remarques sur les tris :

- certains se font en ne manipulant que le tableau et ses éléments (tri *en place*) et d'autres en créant des tableaux auxiliaires,
- on a certains tris qui sont dits *stables* : lorsque 2 données ont la même clé, elles sont encore dans le même ordre après le tri (par exemple pour faire un tri d'abord sur le prénom puis sur le nom pour conserver l'ordre sur les prénoms lorsqu'on a un même nom)
- la complexité maximale est évidemment importante, mais la complexité moyenne prend son intérêt lorsqu'on doit réaliser beaucoup de tris sur des listes assez quelconques
- certains tris sont déterministes (à chaque fois qu'on l'appelle sur un même jeu de données, l'algorithme se déroule de la même façon) ou non-déterministes (ou stochastiques, ou aléatoires) lorsque le déroulement fait apparaître un choix aléatoire (évidemment le résultat est le même en sortie).

I.2 | Quelques tris

Les tris qu'on va présenter :

- **tri par sélection** : on détermine la position du (ou d'un) plus petit élément de la liste et on l'échange avec le premier élément. On recommence avec les $n-1$ derniers éléments...
- **tri bulle** : on fait remonter le plus grand élément par échanges consécutifs puis on recommence avec les $n-1$ premiers éléments...
- **tri par insertion** : on insère les éléments les uns après les autres dans la liste des précédents qu'on a trié
- **tri par comptage** : on sait quelles sont les données triées qu'on peut obtenir - on va simplement compter combien de fois elles apparaissent
- **tri fusion** : on sépare en 2 listes, on les trie et on les fusionne
- **tri rapide - quick sort** : on choisit un élément, on sépare en deux listes (les inférieurs et les supérieurs) qu'on trie séparément puis qu'on concatène.

II | Les tris simples

Dans tous ces algorithmes, on a une liste `L` de taille `n` d'éléments à trier (en général des entiers, des flottants, des chaînes de caractères).

II.1 | Tri par sélection

Principe

- On parcourt toute la liste pour déterminer la position k du minimum (ou d'un minimum s'il y en a plusieurs),
- on échange les éléments en position k et celui en début de liste,
- on recommence avec la liste des $n-1$ derniers éléments.

Remarque : on peut évidemment faire avec le maximum et échanger avec le dernier élément.

Tri par sélection

Données : L une liste à trier
Sorties : vide, la liste L est triée sur place
début
  $n \leftarrow len(L)$
  pour i *de* 0 *à* $n-2$ **faire**
    $p \leftarrow i$
    pour j *de* $i+1$ *à* $n-1$ **faire**
      si *L[j]<L[p]* **alors** $p \leftarrow j$
    $L[p] \leftrightarrow L[i]$

```
def tri_select(L):
    n = len(L)
    for i in range(n-1):
        p = i
        for j in range(i+1,n):
            if L[j]<L[p]: p=j
        L[i],L[p] = L[p],L[i]
```

Commentaires, preuve et complexité

- pas très difficile... à savoir refaire rapidement (bien réfléchir sur les bornes des boucles)
- pour $i = 0$, on a j qui va de 1 à $n-1$ et il y a $n-1$ comparaisons, puis $n-2$ lorsque $i = 1$... Le nombre total de comparaisons est donc $\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$ quelle que soit la liste
- le tri n'est pas stable. Par exemple la liste $[1_1, 1_2, 0]$ est triée en $[0, 1_2, 1_1]$
- pour la preuve, on doit prouver deux invariants, l'un pour la boucle externe et l'autre pour la boucle interne. Pour la boucle externe, on peut prendre, à la fin du passage i :

  $$\text{« } L[0] \leqslant L[1] \cdots \leqslant L[i-1] \leqslant L[i], L[i+1], \ldots, L[n-1] \text{ »}$$

  c'est-à-dire que les i premiers éléments sont inférieurs aux suivants et ils sont triés. Pour la boucle intérieure, on a « pour tout $k \in [\![i\,;j]\!]$, $L[p] \leqslant L[k]$ (le terme en position p est le minimum des termes entre i et j).

II.2 | Tri bulle

Celui-ci on peut l'oublier!!! il n'est là que pour les idées de la partie complexité.

Principe

- On fait remonter le plus grand élément en échangeant $L[i]$ et $L[i+1]$ s'ils sont inversés avec i allant de 0 à $n-2$,
- on recommence avec la liste des $n-1$ premiers éléments.

Tri bulle

Données : L une liste à trier
Sorties : vide, la liste L est triée sur place
début
  $n \leftarrow len(L)$
  pour i *de* 1 *à* $n-1$ **faire**
    $p \leftarrow i$
    pour j *de* 0 *à* $n-1-i$ **faire**
      si *L[j+1]<L[j]* **alors** $L[j+1] \leftrightarrow L[j]$

```
def tri_bulle(L):
    n = len(L)
    for i in range(1,n):
        for j in range(0,n-i):
            if L[j+1]<L[j]: L[j],L[j+1] = L[j+1],L[j]
```

Commentaires, preuve et complexité

- de nouveau on a toujours $\frac{n(n-1)}{2}$ comparaisons, en revanche on a beaucoup plus d'échanges que pour le tri par sélection
- le tri est stable
- Pour compter le nombre d'échanges : on utilise la notion d'inversion (comme pour les permutations) :
  - → le couple (i,j) présente une inversion lorsque $i<j$ et $L[i]>L[j]$.
  - → en échangeant deux éléments *consécutifs*, on supprime une et une seule inversion. À la fin la liste est triée après k échanges ainsi k est le nombre d'inversions de la liste de départ,
  - → on peut calculer le nombre moyen d'inversions sur l'ensemble des permutations des entiers de 1 à n : si σ est une permutation avec k inversions, alors la permutation symétrique σ' possède le nombre complémentaire d'inversions à savoir $\frac{n(n-1)}{2}-k$ (le terme $\frac{n(n-1)}{2}$ est le nombre de couples (i,j) avec $i<j$)). En regroupant les permutations 2 par 2 de cette manière, on a

$$\frac{1}{n!}\sum_{\sigma\in\mathfrak{S}_n} Inv(\sigma) = \underbrace{\frac{1}{n!}}_{\text{poids d'une permutation}} \underbrace{\frac{n!}{2}}_{\text{nombre de bicouples}} \underbrace{\frac{n(n-1)}{2}}_{\text{somme des nombres inversions sur un bicouple}} = \frac{n(n-1)}{4}.$$

- pour la preuve... trouver les invariants de boucles

II.3 | Tri par insertion

Principe

- On commence avec le premier élément,
- on regarde le second élément et on le descend jusqu'à sa place,
- on continue jusqu'au dernier élément.

On va programmer une version « en place ».
Par exempole, sur la liste suivante, les 4 premiers éléments ont été triés. On s'intéresse au cinquième qui vaut 4.

1	8	11	15	4	9	11	17

1	8	11	4	15	9	11	17
1	8	4	11	15	9	11	17
1	4	8	11	15	9	11	17

En pratique, on le réalise plutôt comme cela : on stocke sa valeur dans une variable, on fait remonter d'un cran les éléments précédents, tant qu'ils valent plus que 4 :

1	8	11	15 →	15	9	11	17
1	8	11 →	11	15	9	11	17
1	8 →	8	11	15	9	11	17

et enfin on place l'élément conservé au bon endroit

1	4	8	11	15	9	11	17

Algorithme

```
def tri_insert_n(L):
    n = len(L)
    for i in range(1,n):          # position de l'élément à insérer
        x = L[i]                  # on stocke l'élément
        j = i-1                   # élément à gauche
        while j>=0 and L[j]>x:    # tant que l'élément est supérieur
            L[j+1]=L[j]           # on le décale
            j = j-1               # on passe au précédent
        L[j+1] = x                # on place x au bon endroit
```

Commentaires, preuve et complexité

- le nombre maximal de comparaisons est $\frac{n(n-1)}{2}$ (liste triée à l'envers) et le minimal est $n-1$ lorsque la liste est déjà triée.
- Le nombre moyen semble plus difficile à obtenir : on remarque qu'à chaque fois qu'on fait descendre l'élément à placer d'un cran, on diminue de 1 le nombre d'inversions. Le nombre total de décalage à faire est donc le nombre d'inversion de la permutation. Son nombre moyen est donc équivalent à $\frac{n^2}{4}$ (voir tri bulle).

II.4 | Tri par comptage

Dès qu'on a des informations sur les données qu'on veut trier, on peut proposer des algorithmes plus spécifiques. Ici, on sait que les données sont des entiers compris entre 2 valeurs `v_min` et `v_max`. Plutôt que trier la liste d'entiers `L`, on va simplement compter le nombre d'apparitions de chacun des entiers :

Tri par comptage

Données : `L` une liste à trier, `v_min`, `v_max` les valeurs extrêmes
Sorties : une nouvelle liste triée `T`
début
  `// Initialisations`
  `n` ← `v_max-v_min`
  `C` ← `[0,...,0]` ($n+1$ cases)
  `// Comptage`
  pour *chaque terme* `x` *de* `L` **faire**
    incrémenter la valeur `C[x-v_min]`
  `// reconstruction`
  pour `i` *de* 0 *à* n **faire**
    ajouter `C[i]` termes `v_min+i` dans `T`
  retourner T

Évidemment les valeurs minimales et maximales peuvent également être calculées par la fonction. La complexité temporelle est en $\theta(n+p)$ où p est la longueur de la plage d'entiers et n la taille de la liste à trier. La différence avec les autres tris est qu'il y a besoin de créer un tableau de taille p pour pouvoir compter ce qui peut être coûteux en espace mémoire. On peut à la place utiliser un dictionnaire qui ne ferait alors apparaître que les valeurs distinctes des éléments de `L`. Puisqu'il y en a moins que n, on a alors une complexité en $\theta(n)$.

III | Les tris évolués

Ces deux tris rentrent dans la catégorie des algorithmes « diviser pour régner » (ou « divide and conquer ») : on scinde les données en deux morceaux (si possible les plus équilibrés possibles mais ce n'est pas aussi simple) pour avoir des listes bien plus petites.

IV | Tri fusion

Cet algorithme est important pour le principe de la fusion.

IV.1 | Principe

- on scinde la liste L en deux sous listes L_1 et L_2 (au mileu),
- on trie les deux listes L_1 et L_2 (récursivement avec la conditon d'arrêt lorsque la liste comporte au plus un élément),
- on fusionne les deux listes : on dispose d'un compteur de position pour chacune des listes et on a simplement à comparer chacun des éléments en ces positions (on peut voir cela comme deux piles triées - à chaque étape on regarde les deux sommets en prenant le plus petit des 2 piles).

Algorithme

On présente tout d'abord la fusion de deux listes triées (c'est un présupposé) :

```
def fusion(L1,L2):
    n = len(L1)
    m = len(L2)
    L = [0]*(m+n)
    i, j = 0, 0
    for k in range(m+n):
        if i==n:           # on a épuisé la liste L1
            L[k] = L2[j]
            j += 1
        elif j==m:         # on a épuisé la liste L2
            L[k] = L1[i]
            i += 1
        elif L1[i]<L2[j]: # les deux listes sont 'non vidées'
            L[k] = L1[i]  # on choisit le plus petit des 2 sommets
            i += 1
        else:
            L[k] = L2[j]
            j += 1
    return(L)
```

l'algorithme complet serait alors

```
def tri_fusion(L):
    if len(L)<=1: return L
    n = len(L)
    L1 = tri_fusion(L[:n//2])
    L2 = tri_fusion(L[n//2:])
    return fusion(L1,L2)
```

Complexité

Le fait de couper en 2 récursivement nous incite pour commencer à s'intéresser au cas où n est une puissance de 2. On note C_n la complexité pour trier une liste de n éléments (nombre de tests et d'affectations par exemple). On obtient alors $C_{2^{k+1}} \leqslant 2C_{2^k} + A2^{k+1}$: le $A.2^{k+1}$ correspond à la fusion des deux sous-listes qui est linéaire en le nombre total de termes. Si on note $u_k = \dfrac{C_{2^k}}{2^k}$, on obtient, en divisant la relation précédente par 2^{k+1}, $u_{k+1} \leqslant u_k + A$ ce qui donne $u_k \leqslant u_0 + Ak$ et ainsi $C_{2^k} = O(k2^k)$ ou encore $C_n = O(n\log n)$ lorsque $n = 2^k$. Pour n quelconque, on peut encadre $2^{k-1} < n \leqslant 2^k$ et se convaincre raisonnablement que $C_n \leqslant C_{2^k}$ (par exemple on peut compléter la liste en ajoutant des termes plus grands que tous les autres au bout pour se ramener à une longueur 2^k). On obtient alors $C_n = O(n\log n)$ dans tous les cas.

Amélioration

Pour éviter de passer du temps à recopier des listes notamment lors de la phase de séparation, on peut essayer de réaliser un algorithme modifiant la liste (mais toujours avec une liste intermédiaire dans la fonction) en spécifiant non pas les deux sous listes mais la liste complète et trois entiers $a \leqslant c < b$ qui désignent les extrémités de la liste à traiter et le point de séparation - on crée une liste temporaire pour y placer les éléments triés qu'on recopie ensuite dans L entre les positions a et b... à faire en TD.

IV.2 | Tri rapide

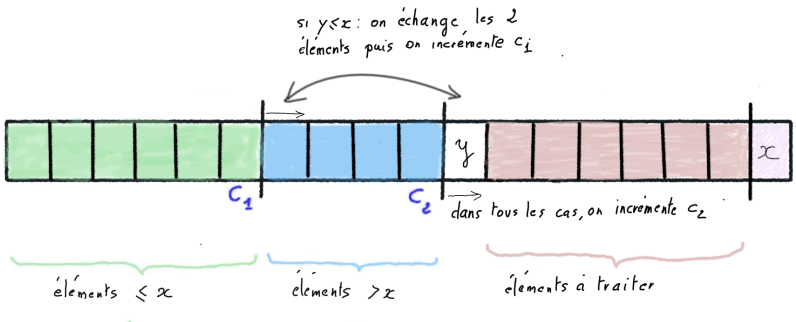
Principe

C'est de nouveau un algorithme récursif qui à l'avantage de pouvoir se faire en place.

- on choisit un élément pivot x dans la liste,
- on sépare la liste en `L1` celle des éléments strictement inférieurs et `L2` contenant ceux qui sont supérieurs ou égaux (on exclut le x choisi afin d'être certain d'avoir des sous-listes de taille strictement inférieure),
- on trie ces 2 sous-listes et on concatène `L1,x,L2`

L'un des avantages est qu'on peut facilement en faire une version en place. La partie plus difficile est de réaliser la séparation en trois parties de la liste : les éléments inférieurs ou égaux à x, le x pivot et ceux qui sont supérieurs. Pour cela

- On échange l'élément pivot x avec le dernier élément,
- on parcourt les termes de la liste en gérant 3 zones consécutives : les éléments explorés inférieurs ou égaux à x, ceux supérieurs et ceux non vus.
- Pour chaque nouvel élément on le place dans l'une des 2 premières zones avec cette méthode :



- finalement on échange x et le premier élément de la seconde zone.

La partition étant réalisée, il n'y a plus qu'à recommencer récursivement sur les deux zones (en gérant des compteurs de début/fin pour délimiter la zone à trier).

Commentaires, complexité

- la première question est de savoir comment choisir le pivot : l'élément de début, de fin, central ou un élément aléatoire. Si les listes sont totalement aléatoires, le fait de choisir le premier ou le dernier (ou le milieu) donnera toujours le même phénomène déterministe avec une complexité du même ordre - de plus lors d'une liste « mal organisée », on peut avoir le risque d'avoir des sous-listes totalement disproportionnées (par exemple si la liste est déjà triée et qu'on choisit l'élément de tête pour trier, on perd le bénéfice du « diviser pour régner »). On peut alors « randomiser » l'algorithme en choisissant un pivot aléatoire. De plus sur une même liste la complexité sera différente à chaque appel avec plus de chance d'obtenir la complexité moyenne.
- On voit assez facilement que la complexité dans le pire des cas est en $O(n^2)$. On montre beaucoup plus difficilement que la complexité moyenne est en $O(n \log n)$.

Chapitre 6 | Graphes

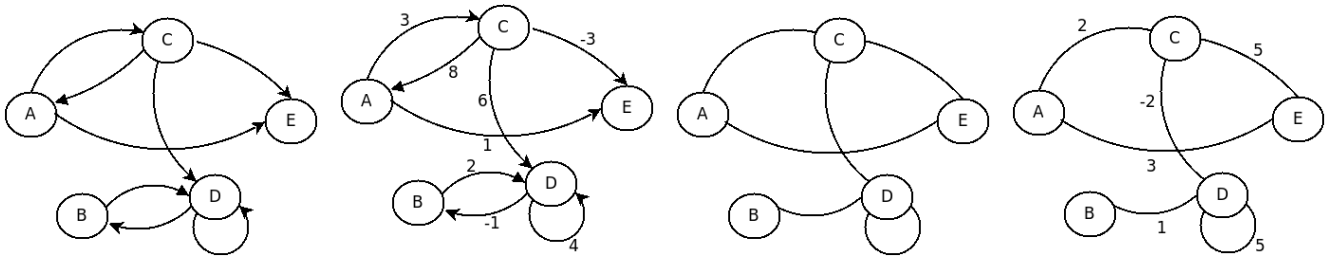
I | Définitions, vocabulaire

Définition *(Graphes)*

Plusieurs notions au programme :

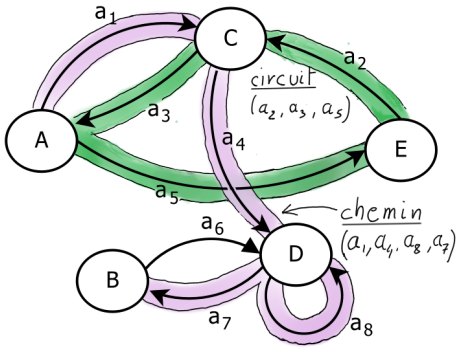
- ***graphe orienté*** : un couple de deux ensembles $G = (S, A)$ où S est un ensemble $\{s_1, \dots, s_n\}$ (les sommets) et $A \subset S \times S$ un ensemble de couples d'éléments de S, $A = \{a_1, \dots, a_p\}$ avec $a_i = (s, s')$ où $s, s' \in S$ (les arcs - elles sont orientées dans le sens $s \leftarrow s'$)
- ***graphe non-orienté*** : un couple de deux ensembles $G = (S, A)$ où S est un ensemble $\{s_1, \dots, s_n\}$ (les sommets) et $A \subset S \times S$, $A = \{a_1, ldots, a_p\}$ avec $a_i = \{s, s'\}$ où $s, s' \in S$ (les arêtes - elles n'ont pas d'orientation)
- ***graphe pondéré*** : ajout d'une application de poids sur les arcs ou les arêtes, c'est-à-dire une fonction $w : S \to \mathbb{R}$

Différents graphes



Graphe orienté | Graphe orienté pondéré | Graphe non orienté | Graphe non orienté pondéré

graphe orienté	graphe non orienté
arc (s, s') : couple, l'ordre est important	arête $\{s, s'\}$: ensemble - l'ordre n'intervient pas
s' **successeur** de s lorsque (s, s') est un arc s' **prédécesseur** de s lorsque (s', s) est un arc	s' **adjacent** à s lorsque $\{s, s'\}$ est une arête
$d_+(s)$: **degré** sortant de s = nombre d'arcs (s, s') $d_-(s)$: **degré** entrant de s = nombre d'arcs (s', s) $d(s) = d_+(s) + d_-(s)$: **degré** du sommet	$d(s)$: **degré** de s = nombre d'arêtes contenant le sommet s
chemin $(a_1, \dots, a_p)$: p-uplet d'arcs tels que l'extrémité terminale de a_i et l'extrémité initiale de a_{i+1}	**chaîne** $(a_1, \dots, a_p)$: p-uplet d'arêtes tels que les arêtes a_i et a_{i+1} aient une extrémité commune
circuit : chemin tel que l'extrémité initiale du premier arc est l'extrémité terminale du dernier arc	**cycle** : chaîne dont les arêtes extrêmes ont un sommet en commun



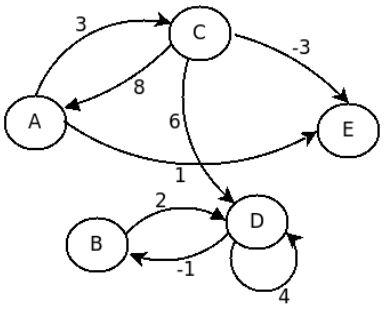
II | Représentation et utilisation des graphes en Python

II.1 | Matrice d'adjacence

On numérote les sommets de 0 à $n-1$. On définit la matrice d'adjacence A avec $a_{ij} = 1$ lorsqu'il y a un arc/une arête entre les sommets i et j

- la matrice est symétrique si le graphe est non orienté,
- à la place de mettre 1, on peut placer le poids de l'arc/arête pour un graphe pondéré
- *avantages* :
  - → on sait facilement s'il y a un arc/une arête entre 2 sommets,
  - → on gère facilement les poids
  - → on peut obtenir les sommets initiaux/terminaux en regardant sur la ligne/colonne correspondante
- *inconvénients* : déterminer les sommets adjacents d'un sommet nécessite de parcourir toute une ligne/colonne (complexité en $O(n)$)

Un exemple : en numérotant les sommets de 0 à 4 au lieu de A à E :

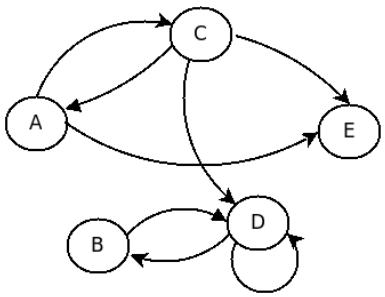


$$\Longleftrightarrow \quad \begin{pmatrix} 0 & 0 & 3 & 0 & 1 \\ 0 & 0 & 0 & 2 & 0 \\ 8 & 0 & 0 & 6 & -3 \\ 0 & -1 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

II.2 | Liste d'adjacence

Pour chaque sommet s , on donne sa liste de sommets adjacents s' (de sorte que (s, s') est un arc ou $\{s, s'\}$ une arête). Cela peut également être réalisé à l'aide d'un dictionnaire, ce qui permet de conserver les noms des sommets.

- on peut gérer les poids en associant à un sommet les couples sommet adjacent/poids **ou** un nouveau dictionnaire avec les sommets en clé et les poids en valeur,
- *avantages* :
  - → on a directement la liste des sommets successeurs (utile par exemple pour tous les algorithmes de parcours) ,
  - → il est plus ou moins facile de savoir si un arc/une arête existe
- *inconvénients* : pas simple d'avoir les sommets prédécesseurs à un sommet dans le cas d'un graphe orienté



$\Longleftrightarrow$

```
Arc = {
'A' : ['C','E'],
'B' : ['D'],
'C' : ['A','D','E'],
'D' : ['B','D'],
'E' : []
}
```

III | Les algorithmes à connaître

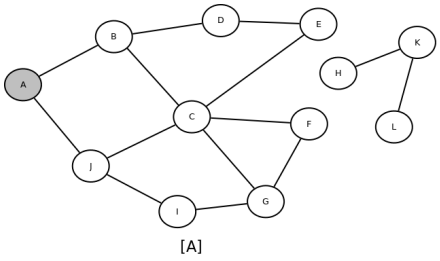
III.1 | Composantes connexes d'un graphe non orienté

Le principe est assez simple : on choisit un sommet s_0, on regarde tous ses sommets adjacents, puis tous les sommets adjacents qu'on n'a pas encore rencontré, et ainsi de suite jusqu'à ne plus avoir de sommet adjacents. On a alors déterminer une composante connexe. On recommence avec un nouveau sommet par encore visité s'il existe. Version algorithme, cela donne

Composante connexe de s dans un graphe

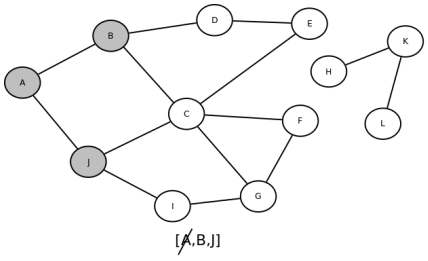
Données : G un graphe, s un sommet
Sorties : la composante connexe L de s dans G
début
  créer une liste L vide
  créer une liste P avec le sommet s
  tant que P *est non vide* **faire**
    choisir un élément a de P, le marquer comme 'vu'
    ajouter a à L
    ajouter tous les successeurs 'non vu' de a dans P
  retourner *L*

Exemple : composante connexe de A dans le graphe



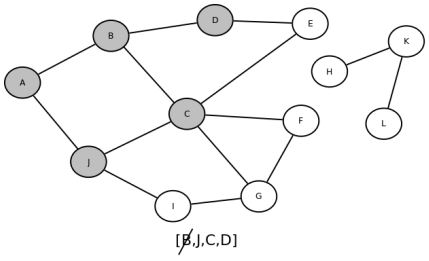
[A]

sommet de départ



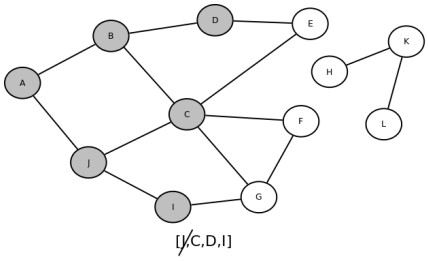
[~~A~~,B,J]

Suppression de A, ajout de ses voisins



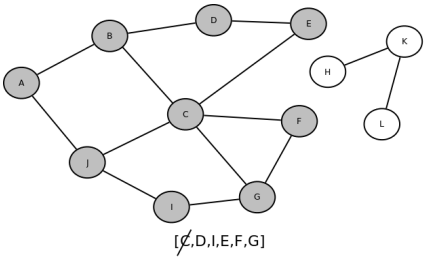
[~~B~~,J,C,D]

Suppression de B, ajout de ses voisins



[~~J~~,C,D,I]

Suppression de J, ajout de ses voisins



[~~C~~,D,I,E,F,G]

Suppression de C, ajout de ses voisins

Pour les derniers sommets dans la pile, il n'y a plus de voisins non utilisés... ils sont supprimés l'un après l'autre jusqu'à avoir une liste vide.

III.2 | Parcours en largeur

C'est exactement le même principe que précédemment. On choisit un sommet, on stocke tous ses voisins (niveau 1). Puis on regarde les nouveaux voisins (niveau 2) de chacun de ces premiers voisins et ainsi de suite jusqu'à ne plus avoir de voisins. On peut à mesure stocker différentes informations : les successeurs d'un sommet, son prédécesseur (un sommet n'en a qu'un seul). On remarque ainsi que cela permet de calcul la distance la plus courte entre le point de départ et les sommets du graphe (on peut même retrouver le chemin en partant de la fin et en remontant dans la liste des prédécesseurs) Évidemment le parcours va dépendre de l'ordre des successeurs d'un sommet (notamment pour l'arbre de parcours obtenu)

GESTION D'UN ENSEMBLE DE VALEURS

On a fréquemment besoin de gérer un ensemble d'objets - par exemple, dans la suite, on utilisera souvent l'ensemble des sommets déjà vu ou marqué avec un certain type. On veut une structure qui permette de manipuler ce genre d'ensemble, c'est-à-dire avec unicité des éléments, si possible un test d'appartenance rapide.

- Un type `set` existe en Python mais n'est pas au programme.
- On pourrait gérer une liste et insérer un élément lorsqu'il n'appartient pas à cette liste. Le problèmes est que le test d'appartenance est linéaire en la taille de la liste donc de plus en plus long lorsque l'ensemble s'agrandit.
- La structure de dictionnaire reste assez adaptée. On utilise les clés pour les objets - avec une valeur quelconque (1 par exemple). L'avantage est que le test d'appartenance est rapide (ainsi que l'ajout - du moins en complexité amortie).

Si de plus on a besoin d'affecter des valeurs aux objets (pour désigner différent statut), la strcuture de dictionnaire s'impose d'autant plus.

Un exemple de programme dans lequel on détermine, par un parcours en largeur, les distances au sommet d'origine, les successeurs d'un sommet dans ce parcours et le prédécesseur (unique) de chacun des sommets. Ce programme fonctionne aussi bien sur un graphe orienté que sur un graphe non orienté

```
def parcours_largeur(G,s):
    statut = {}              # statut d'un sommet (0 : non vu, 1 : vu)
    successeurs = {}         # dictionnaire des successeurs des sommets
    predecesseurs = {}       # dictionnaire du prédécesseur des sommets
    distance = {}            # distance au sommet d'origine

    # Initialisations
    for cle in G.keys():
        statut[cle] = 0
    P = [s]                  # Pile des sommets en cours d'exploration
    predecesseurs[s]=''
    distance[s] = 0
    while len(P)>0:          # tant qu'on a des sommets pour explorer
        sommet = P.pop(0)    # on prend le premier sommet qui fut empilé
        statut[sommet] = 1
        voisins = [ x for x in G[sommet] if statut[x]==0]
                             # les nouveaux sommets voisins
        successeurs[sommet] = voisins
        for v in voisins:
            statut[v] = 1
            predecesseurs[v] = sommet
            distance[v] = 1 + distance[sommet]
        P += voisins         # on empile les nouveaux voisins
    return successeurs, predecesseurs, distance
```

En l'appliquant sur le graphe non orienté suivant à partir du sommet `A`

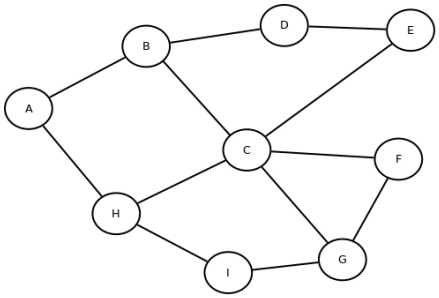
On obtient

```
>>> parcours_largeur(G_no,'A')
# successeurs
{'A': ['B', 'H'], 'B': ['C', 'D'], 'H': ['I'], 'C': ['E', 'F', 'G'], 'D':
    [], 'I': [], 'E': [], 'F': [], 'G': []}
# predecesseurs
{'A': '', 'B': 'A', 'H': 'A', 'C': 'B', 'D': 'B', 'I': 'H', 'E': 'C', 'F': '
    C', 'G': 'C'}
# distances
{'A': 0, 'B': 1, 'H': 1, 'C': 2, 'D': 2, 'I': 2, 'E': 3, 'F': 3, 'G': 3})
```

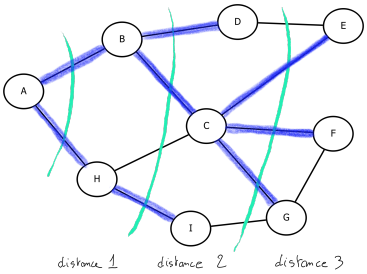
On peut recréer facilement le chemin parcouru à partir du dictionnaire des prédécesseurs :

```
def parcours(sommet,pred):
```

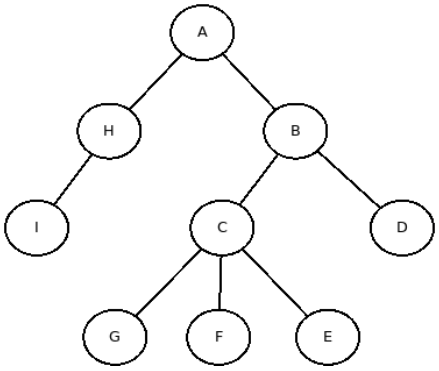
Exemple : parcours en largeur d'un graphe



Graphe d'origine



Parcours et distances



Graphe du parcours

```
    """
    création du chemin entre un sommet (final) à partir du dictionnaire des
        prédécesseurs
    """
    if pred[sommet] == '':
        return sommet
    else:
        return parcours(pred[sommet],pred)+sommet
```

```
>>> parcours('F',P1)
'ABCF'
```

ATTENTION

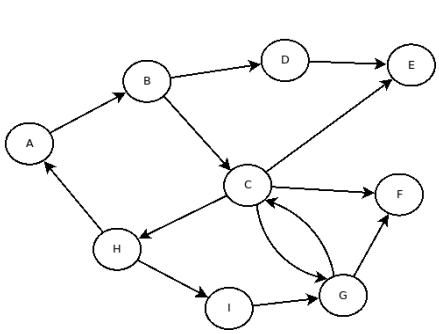
Utilisation d'une file On a ici utilisé une liste sur laquelle on retire le premier élément (`P.pop(0)`). La complexité de cette opération est en $O(n)$ si n est la taille de la liste. On ajoute également des éléments en fin de liste. La structure intéressante pour réaliser cette opération est celle de **file** (en utilisant par exemple les `deque` du modules `collections`).

III.3 | Parcours en profondeur

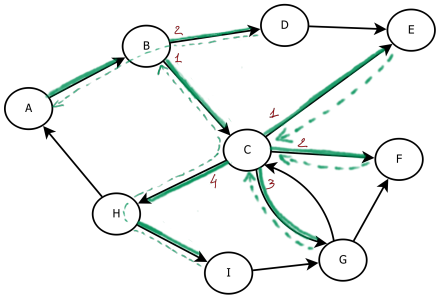
Le parcours se fait différemment cette fois : en partant d'un sommet, on essaie d'aller le plus loin possible dans le graphe, avant de revenir sur nos pas pour poursuivre l'exploration. Contrairement au parcours en largeur où la bonne structure pour les sommets est une file (le premier sommet stocké sera le nouveau point de départ), on utilise cette fois une pile qui va décrire le parcours actuellement réalisé (le bas de la pile est le sommet de départ, le haut le sommet en cours). On fera également attention de ne pas boucler (c'est-à-dire de ne pas repasser par un sommet déjà exploré ou en cours d'exploration).

```
def parcours_profondeur(G,s):
    successeurs = {}
    predecesseurs = {}
    statut = {}
    # 0 : non vu
    # 1 : en cours sur le chemin
    # 2 : épuisé, plus de successeur
    for cle in G.keys():
        statut[cle] = 0
        successeurs[cle]=[]
    P = [s]                                          # 'pile' d'exploration actuelle
    predecesseurs[s]=''
    while len(P)>0:
        sommet = P[len(P)-1]
        statut[sommet] = 1
        voisins = [ x for x in G[sommet] if statut[x]==0]
        if len(voisins)>0:                           # si au moins un successeur
            nouveau = voisins[0]                     # on prend choisit le premier
            successeurs[sommet].append(nouveau)      # qu'on ajoute aux successeurs
                                                     # du sommet actuel
            predecesseurs[nouveau] = sommet          # en stockant le prédécesseur
            sommet = nouveau                         # du nouveau sommet
            P.append(sommet)                         # qu'on ajoute à l'exploration
        else:
            P.pop()                                  # sinon bout de file, on le retire
            statut[sommet]=2                         # et on le passe en définitivement
                vu
    return successeurs,predecesseurs
```

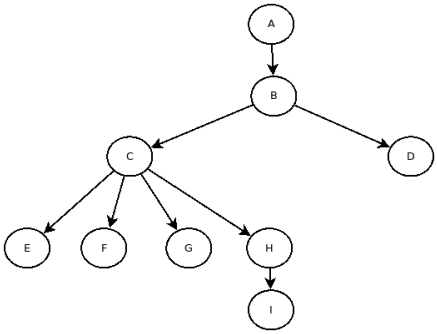
Exemple : parcours en largeur d'un graphe



Graphe d'origine



Parcours et distances



Graphe du parcours

DÉTECTION DE CYCLES

L'algorithme de parcours en profondeur permet, pour des graphes non orientés (c'est ce qui est marqué au programme), de détecter certains cycles. Si on reprend l'exemple du paragraphe précédent, on va créer une chaîne passant par les sommets `A`, `B`, `C`, `E`, `D`. On se rend compte que `D` n'a pas de voisin non marqué mais a un voisin marqué dans le chemin (les éléments de statut 1 dans le programme précédent), à savoir `B`. On en déduit l'existence d'un cycle `BCDEB`. Évidemment on ne détectera pas tous les cycles par ce moyen.

IV | Algorithmes sur les graphes pondérés

Les algorithmes sur les graphes pondérés au programme sont des algorithmes de calcul de plus court chemin

IV.1 | Algorithme de Dijkstra

L'algorithme de Dijkstra permet de calculer les distances (les plus courtes) entre un sommet d'un graphe pondéré (orienté ou non) et tous les autres sommets (connexes). La contrainte est que les poids des arcs/arêtes doivent être positifs ou nuls. Le principe est de créer des sous-graphes qui ont cette contrainte de minimalité, d'ajouter le sommets voisins à un certain sommet bien choisi, puis de mettre à jour les distances sur les sommets voisins. Plus précisément, pour la version simple de l'algorithme :

Algorithme de Dijkstra

Données : `G` un graphe pondéré, `w` les poids, `s` un sommet
Sorties : les distances de chaque sommet à `s`
début
  Associer à chaque sommet une distance infinie dans un objet `D`
  Associer au sommet `s` la distance 0
  Créer un ensemble vide `S`, pour les sommets explorés
  tant que *il reste des sommets non explorés* **faire**
    choisir un sommet `s_act` non exploré de distance `D(s_act)` minimale
    marquer ce sommet comme exploré
    pour *chaque sommet voisin* `s'` *de* `s_act` *non marqué* **faire**
      ajuster sa distance : minimum entre l'actuelle `D(s')` et `D(s_act)+w(s_act,s')`
  retourner *D*

On peut facilement améliorer cet algorithme en créant en même temps les chemins de longueur minimale qui relient l'origine aux sommets : à chaque fois qu'une distance d'un sommet `s'` est mise à jour, il suffit de signaler que le prédécesseur actuel de ce sommet est `s_act`

Le principe est relativement simple. On a un algorithme glouton : à chaque étape on choisit la situation la plus favorable et cela débouche bien sur la situation optimale (cela se démontre - voir cours de première année). La question de la complexité est plus compliquée. Notamment à cause du choix du sommet de distance minimale parmi les sommets restants. Un algorithme basique va être linéaire par rapport au nombre de sommets restants mais on se rend compte que s'il y a peu de sommet modifiés à une étape (sur les distances), on doit pouvoir faire bien mieux que reparcourir l'ensemble de tous les sommets - seuls les sommets mis à jour ainsi que celui de distance minimale dans ceux non mis à jour sont intéressants. On peut implémenter cela à l'aide d'une file de priorité (mais on n'en dira pas plus ici).

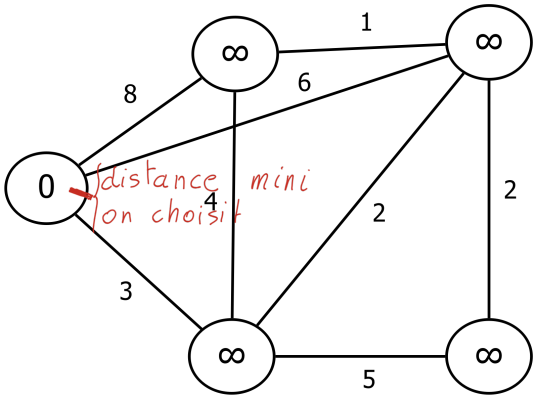
Comme signalé précédemment, l'algorithme est également valable sur un graphe orienté (du moment que les arcs ont un poids positif ou nul).

L'INFINI

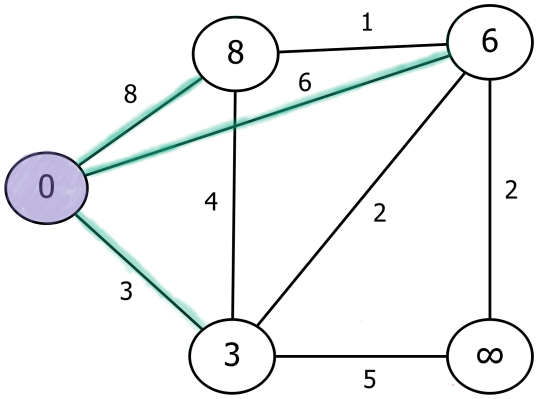
Bien que n'étant pas officiellement au programme, il existe un infini en Python, obtenu avec la conversion en flottant de la chaîne "inf" : `float('inf')`. On peut lui affecter un signe et faire les opérations usuelles avec :

```
>>> a = float('inf')
>>> -3*a
-inf
>>> a+2
inf
>>> 4 <= a
True
>>> a-a
nan # not a number
```

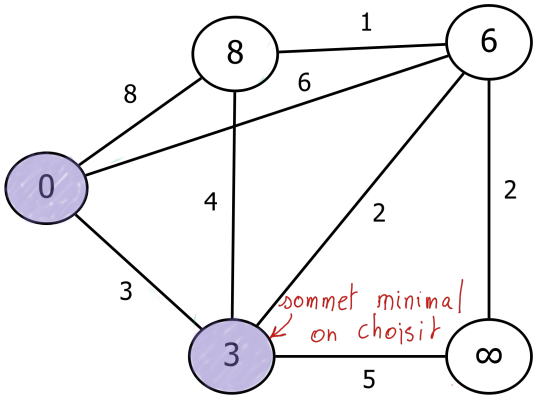
Exemple : algorithme de Dijkstra



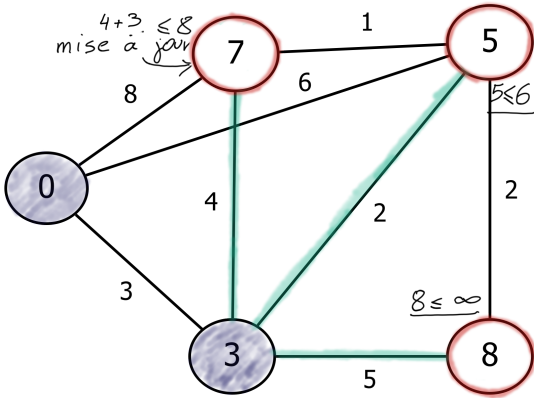
Situation d'origine



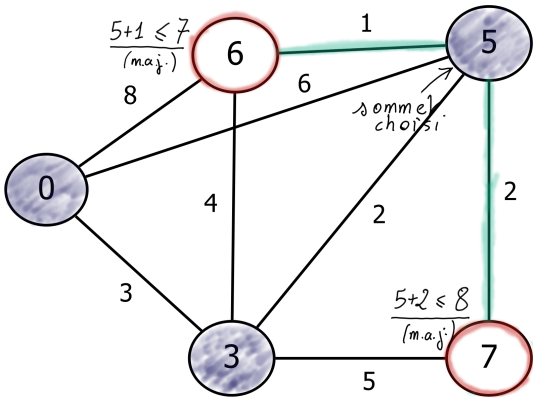
mise à jour des distances



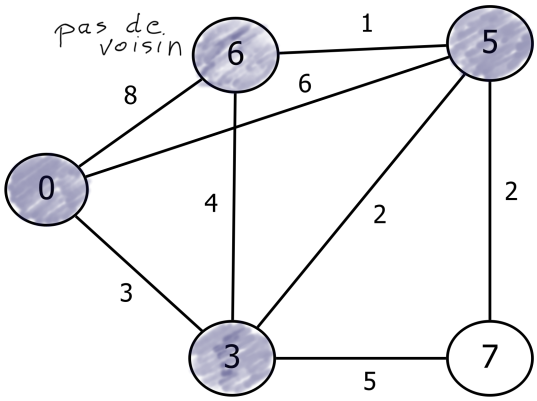
Choix du prochain sommet



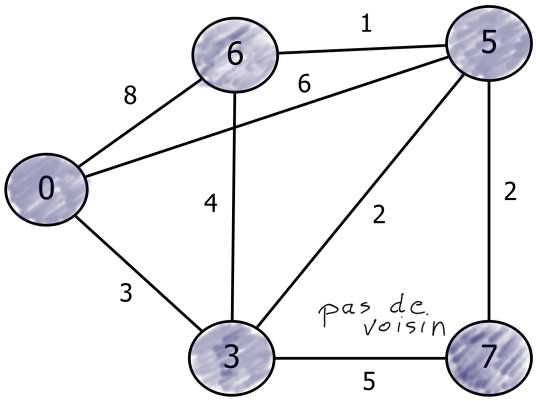
Mise à jour des distances des voisins (non marqués)



Choix du sommet suivant et maj des voisins



pas de voisin



Dernier sommet - pas de voisin

On peut comme signalé ajouter un dictionnaire avec les prédécesseurs : à chaque fois que la nouvelle distance est inférieure à l'actuelle, on change le prédécesseur du sommet mis à jour.

Une implémentation possible (avec une gestion simple du choix du minimum) :

```
def dijkstra(G,s):
    """
    algorithme de Dijkstra sur un graphe pondéré G, à partir du sommet s
    retour : dictionnaire avec les distances
    """

    def sommet_mini():  # clé du sommet de distance mini parmi les non marqués
        L = [cle for cle in G.keys() if statut[cle]==0]
        mini = float('inf'); cle = ''
        for k in L:
            if D[k]<mini:
                mini = D[k]
                cle = k
        return cle       # " si la liste L est vide

    # initialisations
    D = {} ; statut = {}
    for sommet in G.keys():
        D[sommet] = float('inf')     # c'est l'infini en Python
        statut[sommet] = 0
    D[s] = 0
    s_act = s                        # sommet de départ
    while s_act:                     # tant qu'on a un sommet minimal
        statut[s_act] = 1            # le sommet actuel devient vu
        voisins = [k for k in G[s_act].keys() if statut[k]==0]
        # on choisit les voisins non vu du sommet actuel
        for v in voisins:
            D[v] = min(D[v], D[s_act]+G[s_act][v])
            # pour chacun de ces sommets, on met la distance à jour
        s_act = sommet_mini()
        # nouveau sommet 'actuel'
    return D
```

IV.2 | Algorithme A*

Imaginons un graphe assez grand, par exemple un graphe routier avec plusieurs centaines/milliers de routes. On souhaite aller d'une ville à une autre par le chemin le plus court. L'algorithme de Dijkstra permet cela mais il va explorer les sommets dans un ordre non adapté à ce problème. On imagine que le chemin le plus court est proche de la « ligne droite » entre ces deux points. On va étudier un nouvel algorithme proche de l'algorithme de Dijkstra avec une différence essentiellement sur le choix du sommet voisin (pour rappel, dans l'algorithme de Dijkstra, on choisissait le sommet non exploré à une distance minimale du sommet de départ). Pour un sommet, au lieu de simplement le choisir en fonction de sa distance actuelle à l'origine comme dans l'algorithme de Dijkstra, on va décomposer son coût en deux :

$$\text{cout}(S) = d(S) + h(S)$$

où

- $d(S)$ représente la distance au sommet d'origine, calculé et mis à jour dans l'algorithme (de la même façon que l'algorithme de Dijkstra)
- $h(S)$ une valeur (appelée heuristique) qui approche la distance de ce sommet au sommet de fin. On a plusieurs choix possible - par exemple, on peut considérer la distance euclidienne (le but est d'avoir une bonne heuristique afin de ne pas surestimer cette valeur, ce qui conduirait à un résultat non optimal).

Le principe est le même que l'algorithme de Dijkstra avec comme différence qu'on choisit un sommet de coût minimal parmi ceux qui ne sont pas exploré (ainsi on aura tendance à d'abord choisir des sommets plus proches de la fin qu'un sommet un peu au hasard).

Chapitre 7 | Programmation dynamique

I | Dictionnaires

Un dictionnaire est un ensemble constitué de couples « clé :valeur ». Il n'est pas ordonné. Les clés sont uniques et à chacune de ces clés est associée une valeur. Les clés doivent être des objets non mutables - en général on utilise des entiers, des chaînes de caractères ou des tuples, les valeurs peuvent être totalement quelconques.

DICTIONNAIRES : CRÉATION ET MANIPULATION

`dict()` ou `{}`	dictionnaire vide
`{c1:v1, c2:v2, ...}`	création d'un dictionnaire avec les clés c_i et les valeurs v_i
`d[cle]`	accès à la valeur associée à la clé `cle` (en lecture et écriture). Lors d'une affectation, la clé est créée si elle n'existe pas
`d.keys()`	ensemble des clés du dictionnaires (objet itérable)
`d.values()`	les valeurs du dictionnaires (objet itérable)
`d.items()`	les couples clé/valeurs du dictionnaires (objet itérable)
`len(d)`	taille du dictionnaire (nombre de clés)
`k in d` (ou `k not in d`)	teste si la clé `k` est présente (absente) dans le dictionnaire
`d.copy()`	réalise une copie superficielle du dictionnaire
`d.pop(cle)`	retire le couple clé/valeur du dictionnaire et renvoie la valeur
`d.popitem()`	retire et renvoie un couple clé/valeur du dictionnaire (le dernier créé)
`del d[cle]`	efface le couple clé/valeur associé à la clé

I.1 | Comparaison listes et dictionnaires

Contrairement à ce que l'on pourrait croire, l'accès aux éléments d'un dictionnaire est très efficace (elle utilise ce qu'on appelle une fonction de « hachage » qui associe un entier à chaque élément du dictionnaire). Les points à retenir (en simplifiant, notamment sur la fonction de hachage) et en notant `n` la taille de la liste/dictionnaire :

- une liste est intéressante lorsque l'ordre des éléments est important, ou que l'indexation par des entiers est importante - à l'opposé, une dictionnaire permet d'avoir un ensemble de clés totalement quelconque,
- le test d'appartenance est en $O(1)$ pour un dictionnaire, alors qu'il peut être en $O(n)$ pour une liste
- les temps d'accès à un élément donné est en $O(1)$ pour les deux (un peu plus rapide pour les listes),
- le temps de suppression d'un élément est en $O(1)$ pour un dictionnaire et en $O(n)$ pour une liste (sauf si c'est le dernier élément qu'on retire avec `pop()`)

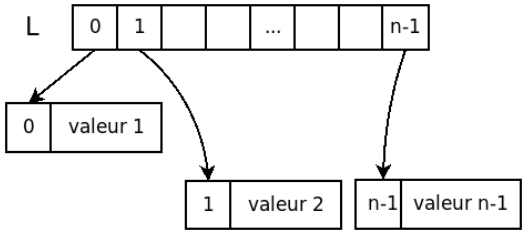
```
>>> d = { 'a':3, 'b':5, 'c':4 }
>>> d['a']
3
>>> d['e']
KeyError
>>> 'b' in d
True
>>> d['e'] = 5 ; d['b']=3*d['e']
>>> d
{'a': 3, 'b': 15, 'c': 4, 'e': 5}
```

On peut avoir des clés qui sont des tuples, mais pas des listes (une clé doit être immuable)

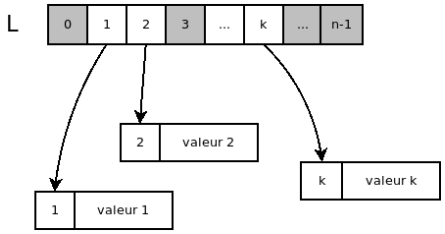
```
>>> d = {} ; d[(0,0)]=1 ; d[(1,0)]=2 ; d[(2,3)] = 8 ; d
{(0, 0): 1, (1, 0): 2, (2, 3): 8}
>>> d[[2,1]] = 0
TypeError: unhashable type: 'list'
```

I.2 | Adressage directe vs hachage

L'un des moyens simples de gérer un ensemble de valeurs est la liste dynamique Python. Elles sont gérées par adressage direct. L'univers des clés est un ensemble d'entiers de 0 à $n-1$ et on a directement accès à la case k, puis à sa donnée correspondante :

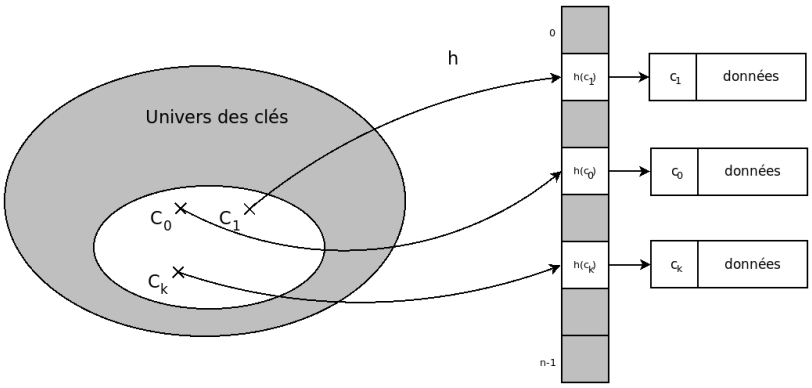


On pourrait alors avoir une structure analogue mais « avec des trous », lorsqu'on a un ensemble fini de clés possibles mais qu'elles ne sont pas toutes utilisées :



On voit l'avantage évident d'une plus grande souplesse mais également l'inconvénient immédiat : si l'ensemble des clés possibles est très grand alors on aura réservé un tableau d'adresse très grand (donc beaucoup de place mémoire) pour finalement peu de cases utilisées. Admettons qu'on veuille des clés qui sont des chaînes de 5 caractères, on aurait 26^5 (presque 12 millions) cases à réserver pour n'en utiliser que quelques centaines/milliers.

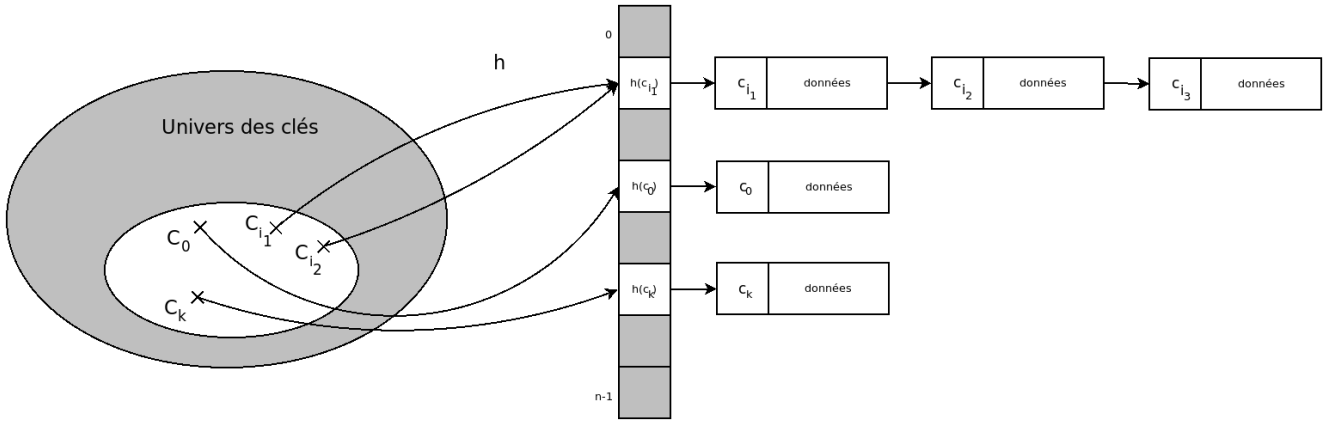
L'idée est donc de réduire l'ensemble des clés par **une fonction de hachage**, c'est-à-dire une fonction h qui part du domaine des clés et renvoie un entier entre 0 et $n-1$ de sorte que pour les clés utilisées, les valeurs de h soient différentes. Cela donne un schéma



Évidemment la fonction h ne sera pas injective... le but est d'avoir une fonction de hachage suffisamment aléatoire pour qu'il n'y ait pas de *collisions* sur les valeurs. Tant que la situation reste sous la format du dessin précédent, le temps nécessaire pour savoir si une clé est présente dans le dictionnaire est en $O(1)$ (on évalue la fonction h sur la clé souhaitée et on regarde s'il y a une valeur correspondante).

Plusieurs questions se posent alors :

- Quelle taille choisir pour les valeurs de h ? l'idée, en Python, est de partir sur une taille pas trop grande et de doubler la taille du dictionnaire dès qu'on se rapproche de cette valeur (comme les fonctions de hachage vont travailler modulo n, c'est encore plus simple si n est une puissance de 2 - ce qui explique le doublement).
- que faire s'il y a malgré tout des collisions ? le principe est de chaîner les données dont les clés ont même valeur hachée, comme dans le dessin suivant. Le test d'existence d'une clé sera un peu plus long puisqu'il faudra parcourir la liste chaînée dans certains cas).



Exercice 7.11 ***(Compter les éléments)***

On dispose d'une liste d'éléments L (liste d'entiers, liste de chaînes de caractères...).

1. Écrire une fonction qui détermine les éléments différents de cette liste et qui, pour chacun d'eux, donne leur nombre.
2. Écrire une fonction qui détermine quel élément apparaît le plus dans une liste (ou l'un de ces éléments s'il y en a plusieurs)
3. Écrire une fonction qui détermine la liste des éléments qui apparaissent le plus (comme la question précédente mais avec une liste cette fois)

II | Programmation dynamique

On aborde une nouvelle méthode de résolution de problèmes. Son principe général repose sur une approche récursive dans laquelle on casse le problème principal en sous-problèmes similaires. Pour illustrer le principe général, on peut parler du problème de la recherche du plus court chemin entre deux sommets A et B. Si ce chemin passe par un sommet C alors le sous-chemin de A vers C est aussi minimal (sinon on aurait un chemin plus court entre A et C, ce qui permettrait de créer un chemin plus court entre A et B). La technique est très différente de celles des algorithmes de type *diviser pour régner* - penser aux exemples d'algorithmes de tri , tri-fusion ou tri rapide - car pour ce dernier type, les sous-problèmes sont plutôt indépendants alors que dans la programmation dynamique, au contraire, on obtient des sous-problèmes avec de fort liens entre eux.

II.1 | Mémoïsation

On commence par exemple vraiment pas original, la suite de Fibonacci. Elle est définie par $F_0 = F_1 = 1$ et, pour tout $n \geqslant 2$, $F_n = F_{n-1} + F_{n-2}$. On peut revoir les commentaires en page 27. La méthode récursive basique recalcule les termes plusieurs fois. Le principe de la mémoïsation est de conserver une valeur déjà calculée afin de la réutiliser si besoin. On évite donc les rappels inutiles à la fonction mais la contrepartie est qu'on doit stocker toutes (ou beaucoup de valeurs) de la fonction.

Voici plusieurs versions de la fonction :

- la première approche est de conserver la définition récursive et de stocker les valeurs qu'on a effectivement calculées

```
Fibo = {}

def fibo_rec(n):
    if n==0 or n==1:
        Fibo[n] = 1
    if n not in Fibo: # si le terme n'a pas déjà été calculé
        Fibo[n] = fibo_rec(n-1)+fibo_rec(n-2)
    return Fibo[n]
```

- la seconde approche est au contraire de partir du début et de remonter jusqu'à la valeur souhaitée

```
def fibo_iter_dict(n):
    Fibo_loc = {0 : 1, 1 : 1 }
```

```
    for i in range(2,n+1):
        Fibo_loc[i] = Fibo_loc[i-1]+Fibo_loc[i-2]
    return Fibo_loc[n]
```

avec un dictionnaire, ou simplement avec une liste :

```
def fibo_iter_liste(n):
    Fibo_loc = [1,1]+[0]*(n-1)
    for i in range(2,n+1):
        Fibo_loc[i] = Fibo_loc[i-1]+Fibo_loc[i-2]
    return Fibo_loc[n]
```

Plusieurs commentaires à faire sur ces exemples :

- on peut évidemment, dans les versions itératives, utiliser une variable `Fibo` globale et continuer à la remplir uniquement si on demande une valeur plus élevée que celles calculées (en plus en reprenant directement au dernier terme calculé)
- la version récursive est toujours limitée par la profondeur de récursion autorisée dans Python. On ne peut pas directement demander F_{10000} alors que la version itérative le permet,
- on a évidemment un gain de temps énorme par rapport à la version récursive basique (on passe d'une complexité exponentielle à une complexité linéaire),
- on voit apparaître deux approches très différentes entre les deux versions :
  - → la version itérative est de type **Bottom-Up** ou **de bas en haut** : on part du bas et on remonte vers les termes d'indices plus élevés,
  - → la version récursive est de type **Top-Down** ou **de haut en bas** : on détermine la valeur d'indice n en redescendant vers les termes d'indices inférieurs (jusqu'à pouvoir les calculer)

II.2 | Rendre la monnaie

Principe

On dispose de pièces de certaines valeurs pour rendre la monnaie (on suppose que le nombre de pièce est illimité). On doit rendre un certain montant à l'aide de ces pièces, en donnant le moins de pièces possibles.

On dispose d'une certaine liste ordonnée de pièces, par exemple pour les pièces usuelles, on a

```
L = [1,2,5,10,20,50,100,200]
```

mais on peut très bien disposer d'autres valeurs (il suffit d'avoir une pièce de 1 pour être certain de pouvoir toujours le faire). Si on veut rendre 134, on peut le faire en le décomposant en $100+20+10+2+2$

Algorithme glouton

Le principe est extrêmement simple : on rend la pièce la plus élevée possible à chaque fois. Cet algorithme ne rend pas toujours la meilleure réponse ; en fait cela dépend du jeu de pièce... on dit que le jeu de pièces est canonique lorsque l'algorithme glouton fournit la meilleure réponse - ce qui est le cas pour les pièces usuelles. Par exemple avec `L=[1,4,6]`, l'algorithme renvoie 3 pièces $(6+1+1)$ alors que le meilleur choix est $4+4$.

Programmation dynamique

Si on doit rendre une valeur de n avec le moins de pièces possibles et si l'une des pièces à la valeur k, alors les $n-k$ restant doivent être rendu de façon optimale (c'est ce qu'on appelle la propriété de **sous-structure optimale**). Si on note t_n le nombre minimum de pièces pour avoir une somme de n, on obtient la relation de récurrence :

$$t_n = 1 + \inf_{c \in L} t_{n-c}.$$

cette borne inférieure peut être atteinte pour différentes valeurs de c (mais ce n'est pas encore la question). On obtient ainsi une formule de récurrence simple permettant de calculer t_n. Afin de ne pas à avoir à recalculer les valeurs de t_{n-c} beaucoup trop de fois, on réutilise la technique de mémoïsation pour stocker les valeurs précédentes de t.

Une version avec une liste recalculée à chaque fois :

```
L = [1,2,5,10,20,50,100,200]

def rendu(n):
    T = [0]*(n+1)
    T[1] = 1
    for i in range(2,n+1):
        mini = n
        for piece in L:
            if i>=piece and T[i-piece]<mini:
                mini = T[i-piece]
        T[i] = 1+mini
    return T
```

donne

```
>>> rendu(10)
[0, 1, 1, 2, 2, 1, 2, 2, 3, 3, 1]
```

On peut également utiliser un dictionnaire qui permet de ne pas refaire les calculs déjà fait pour d'autres valeurs :

```
T = {0 : 0, 1 : 1}
def rendu(n):
    if n not in T:
        mini = n
        for piece in L:
            if n>=piece and rendu(n-piece)<mini:
                mini = rendu(n-piece)
        T[n] = 1+mini
    return T[n]
```

ce qui donne

```
>>> rendu(7)
2
>>> T
{0: 0, 1: 1, 2: 1, 3: 2, 4: 2, 5: 1, 6: 2, 7: 2}
```

Remarque : on retrouve les deux approches « Top-Down » pour la version avec dictionnaire et l'approche « Bottom-Up » pour la version avec une liste. On peut exploiter l'approche de *bas en haut* avec un dictionnaire :

```
def rendu(n):
    k = len(T)  # censé être rempli de 0 à k-1
    for i in range(k,n+1):
        mini = n
        for piece in L:
            if i>=piece and rendu(i-piece)<mini:
                mini = rendu(i-piece)
        T[i] = 1+mini
    return T[n]
```

L'inconvénient de cette méthode est qu'on suppose que le dictionnaire `T` est global et qu'il n'est surtout pas modifié par quelqu'un d'autre ailleurs (il faudrait lui donner un nom plus robuste afin qu'il y ait moins de risque)

Reconstituer le rendu

Savoir combien de pièces rendre, c'est bien. Savoir lesquelles on doit rendre c'est encore mieux!

- l'une des méthodes est de stocker, pour chaque montant, la liste des pièces à rendre et, lors du calcul du minimum, ajouter à cette liste la pièce en cours. Cela risque de créer de longue liste (et ainsi prendre beaucoup de place)
- on peut le faire de nouveau récursivement : pour le montant n, il suffit de savoir qu'elle est l'une des pièces à rendre et de repartir ensuite au montant restant. En terme de graphe (si les sommets sont les montants), cela revient à connaître l'un des prédécesseurs du sommet en question. Il suffit donc soit de connaître le montant précédent, soit la pièce à rendre pour atteindre ce montant.

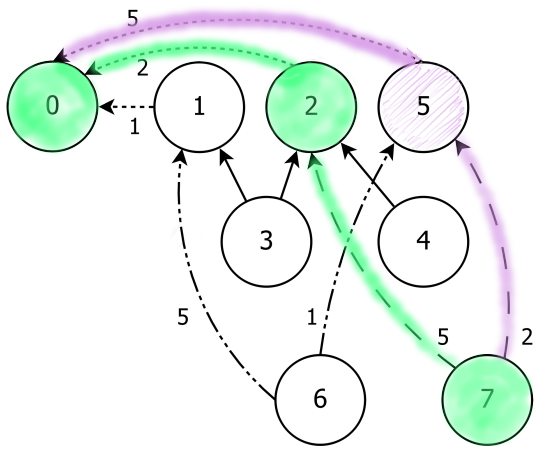


FIGURE 7.1 – deux façons (identiques) pour rendre 7

On peut adapter l'algorithme `rendu` pour qu'il renvoie le nombre de pièces et la dernière pièce rendue, puis écrire une fonction qui donne toutes les pièces :

```
T = {0 : [0,0], 1 : [1,1]}
def rendu(n):
    if n not in T:
        mini = n
        p = 0 # pièce à rendre
        for piece in L:
            if n>=piece and rendu(n-piece)<mini: # rendu doit renvoyer un entier
                mini = rendu(n-piece)
                p = piece
        T[n] = [1+mini,p] # liste avec le nombre et la dernière pièce
    return T[n][0] # on ne renvoie que le montant (raison 4 lignes au dessus)
```

ce qui permet de reconstruire la liste des pièces :

```
def Pieces_a_rendre(n):
    nombre = rendu(n) # crée le dictionnaire complet
    if n==0:
        return []
    else:
        k,p = T[n]
        return Pieces_a_rendre(n-p)+[p]
```

ce qui donne par exemple

```
>>> Pieces_a_rendre(19)
[10, 5, 2, 2]
```

II.3 | Meilleur chemin - déplacements Nord-Est

Présentation du problème

On dispose d'une matrice de taille $p\times(q$ remplies d'entiers positifs (ou de réels positifs). On part de la case en bas à gauche et on veut rejoindre la case en haut à droite par des déplacements uniquement vers le haut ou la droite et en maximisant la somme des valeurs des cases parcourues.

On doit réaliser exactement $p-1$ déplacements vers le haut et $q-1$ vers la gauche. Un tel déplacement peut être représenté par une liste de $p+q-2$ termes N ou E avec exactement $p-1$ qui valent N (ceux vers le haut). Cela donne $\binom{p+q-2}{p-1}$ déplacements possibles. On ne va évidemment par les tester tous. Si on prend $p=q=n=1$, ce nombre de chemins vaut

$$\frac{(2n)!}{(n!)^2} \underset{n\to+\infty}{\sim} \frac{\sqrt{4n\pi}(2n/e)^{2n}}{2n\pi(n/e)^{2n}} \underset{n\to+\infty}{\sim} \frac{4^n}{\sqrt{n\pi}}$$

On va de nouveau utiliser les techniques précédentes, mais cette fois ci sur deux dimensions.
On travaillera sur l'exemple suivant :

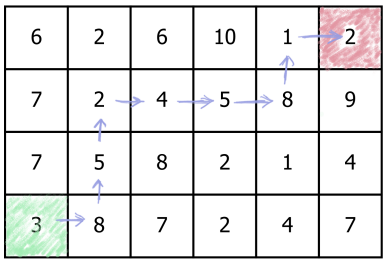
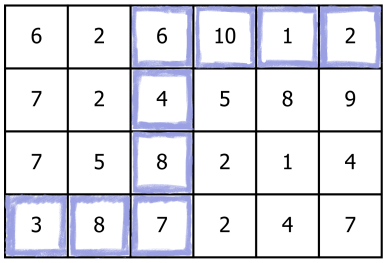


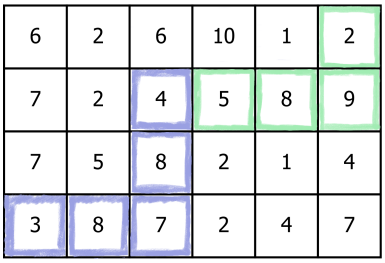
FIGURE 7.2 – Exemple de matrice avec un gain de 38

Algorithme glouton

On part en bas à gauche. À chaque étape, on choisit la situation qui permet de gagner le plus, jusqu'à aboutir à l'arrivée. C'est évidemment facile à programmer, la complexité est en $\theta(p+q)$ (choisir un terme parmi deux à chaque pas) mais cela ne donne pas forcément le résultat optimal



(a) Exemple avec algorithme glouton



(b) Un meilleur chemin

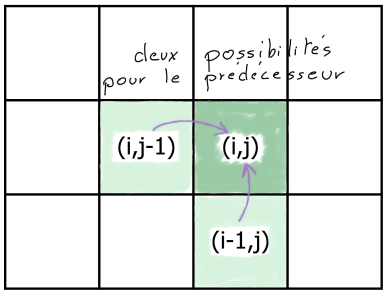
FIGURE 7.3 – Illustration de l'algorithme glouton

Programmation dynamique : calcul du gain maximum

Considérons la case d'arrivée. On y arrive par l'une des deux cases adjacentes. Plus précisément, on aura le gain maximum si et seulement si on arrive de la case qui a elle même un gain maximum à partir du départ. On a donc à connaître le gain maximum des chemins allant à chacune de ses deux cases... et ainsi de suite. Plutôt que de répondre au problème de départ, on va généraliser et déterminer le gain maximum pour chacune des cases du tableau. Si on note $t_{i,j}$ le gain maximum en arrivant à la case (i,j) (et $a_{i,j}$ la valeur de cette case (i,j), on aura alors la relation

$$t_{i,j} = a_{i,j} + \max(t_{i-1,j}, t_{i,j-1})$$

formule illustrée par :



On peut alors proposer un algorithme basé sur ces remarques afin de remplit un tableau avec les gains maximals pour chaque case en remplissant case par case :

- pour les premières ligne et colonne : un seul chemin possible (toujours vers la droite ou vers le haut) donc on commence par affecter ces cases,

- on remplit alors ligne par ligne (ou colonne par colonne si on préfère) à l'aide de la relation de récurrence précédente jusqu'à la dernière case.

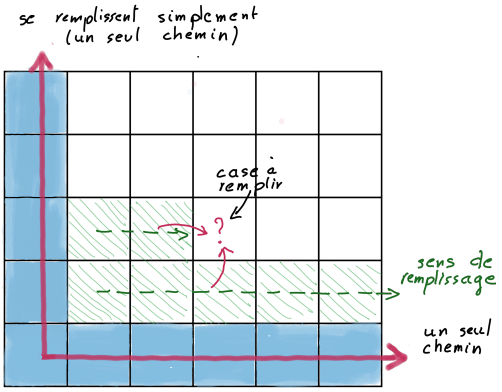


FIGURE 7.4 – Ordre de remplissage des cases

On obtient un algorithme de complexité $O(pq)$

```
\textsc{M = [
[3,8,7,2,4,7],
[7,5,8,2,1,4],
[7,2,4,5,8,9],
[6,2,6,10,1,2]
]}

def gain(M):
    p,q = len(M), len(M[0])
    Gain = [0]*p
    # remplissage ligne 0
    L = [0]*q
    L[0] = M[0][0]
    for c in range(1,q):
        L[c] = L[c-1]+M[0][c]
    Gain[0] = L
    # remplissage des autres lignes
    for ligne in range(1,p):
        L = [Gain[ligne-1][0]+M[ligne][0]]+([0]*(q-1))
        for colonne in range(1,q):
            L[colonne] = M[ligne][colonne] + max(L[colonne-1],Gain[ligne-1][
                colonne])
        Gain[ligne] = L
    return Gain
```

Remarque : les lignes sont dans l'ordre inverse, la ligne 0 est en haut

On peut tester :

```
>>> gain(M)
[[ 3, 11, 18, 20, 24, 31],
 [10, 16, 26, 28, 29, 35],
 [17, 19, 30, 35, 43, 52],
 [23, 25, 36, 46, 47, 54]]
```

Détermination du chemin

Une fois le tableau rempli, on peut reconstituer les chemin inverse en partant de la case terminale : à chaque étape, il suffit de remonter à celle des deux cases possibles qui a le gain le plus élevé. On peut éventuellement , lorsqu'on remplit le tableau des gains, stocker le prédécesseur afin de remonter le chemin facilement.

```
def gain_pred(M):
    p,q = len(M), len(M[0])
    Gain = [0]*p
    Predecesseur = {}
    # remplissage ligne 0
    L = [0]*q
    L[0] = M[0][0]
    Predecesseur[(0,0)] = None
    for c in range(1,q):
        L[c] = L[c-1]+M[0][c]
        Predecesseur[(0,c)] = (0,c-1)
    Gain[0] = L
    # remplissage des autres lignes
    for ligne in range(1,p):
        L = [Gain[ligne-1][0]+M[ligne][0]]+([0]*(q-1))
        Predecesseur[(ligne,0)]=(ligne-1,0)
        for colonne in range(1,q):
            if L[colonne-1]>Gain[ligne-1][colonne]:
                L[colonne] = M[ligne][colonne] + L[colonne-1]
                Predecesseur[(ligne,colonne)] = (ligne, colonne-1)
            else:
                L[colonne] = M[ligne][colonne] + Gain[ligne-1][colonne]
                Predecesseur[(ligne,colonne)] = (ligne-1, colonne)
        Gain[ligne] = L
    return Gain, Predecesseur
```

et pour reconstituer le chemin :

```
def chemin(M):
    p,q = len(M), len(M[0])
    Gain, Pred = gain_pred(M)
    case = (p-1,q-1)
    Chemin = []
    while case!=None:
        Chemin = [case] + Chemin
        case = Pred[case]
    return Chemin
```

ce qui donne sur l'exemple

```
>>> chemin(M)
[(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3), (2, 4), (2, 5), (3, 5)]
```

On n'est pas obligé de créer cet ensemble des prédécesseurs. On pourrait le recalculer de proche en proche une fois qu'on dispose du tableau des gains de chaque case (depuis la case d'origine).

II.4 | Bilan et commentaires

Concepts essentiels pour la programmation dynamique

La programmation dynamique repose sur deux concepts essentiels :

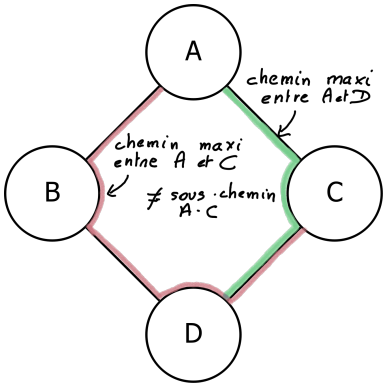
- Sous-structure optimale : une solution au problème contient des solutions des sous-problèmes. Dans le cas du rendu de monnaie, si on prend un sous-montant intermédiaire, la solution obtenue fait notamment apparaître une solution optimale pour le sous-montant. De même, pour la recherche du gain maximum entre les deux extrémités du tableau, un chemin optimal va contenir une solution d'un chemin optimal pour un point intermédiaire. La difficulté consiste, lorsqu'on a un problème, à déterminer quels sont les sous-structure qui vont être utile à la résolution (il ne faut pas

non plus qu'il y en ait trop). Si on reprend la recherche du gain maximum, on aurait pu également essayer de calculer les gains maximal entre deux points quelconques du tableau - s'il sont bien ordonnés. Cela reste valable car pour deux points intermédiaires d'une solution optimale du problème général le chemin proposé est optimal pour ces deux points - sinon on pourrait créer un chemin meilleur pour le chemin global. Ce n'est pas intéressant (et cela complique la programmation) pour cette situation.

- Chevauchement des sous-problèmes : les sous-problèmes ne sont pas indépendants. Les algorithmes récursifs passeraient leur temps à effectuer plusieurs fois des calculs déjà effectués. C'est le cas pour la suite de Fibonacci, pour le rendu de monnaie (rendre 9 en commençant par rendre 5 ou 14 en commençant par 10 amène au même sous-problème. La programmation dynamique est alors un bon choix (par exemple, par mémoïsation).

Peut-on toujours s'y ramener ?

Le point fondamental est donc d'avoir cette propriété de sous-structure optimale. Par exemple, dans un graphe, si on considère le plus court chemin d'un sommet A vers un sommet B et que ce chemin passe par C alors le chemin entre A et C est obligatoirement minimal (sinon en le recollant avec celui de B vers C, on obtiendrait un chemin plus court entre A et B). Ce n'est plus le cas si on considère la recherche du chemin élémentaire (qui ne passe pas deux fois par un même sommet) le plus long. Dans l'exemple qui suit, le chemin ACD est un chemin maximal entre A et D, il passe par le sommet C et le sous-chemin AC n'est pas optimal pour la propriété recherchée puisque le chemin $ABDC$ est plus long.



III | D'autres exemples et exercices

III.1 | Partition équilibrée d'un tableau d'entiers positifs

On dispose d'un,e liste d'entiers positifs `L`. On cherche à partitionner cette liste en deux sous-listes de sorte que les sommes des entiers des sous-listes soient le plus proche possible. Plus précisément, si on note $S(J) = \sum_{x \in J} x$ lorsque $J \subset L$, on cherche L_1 et L_2 telles que

- $L_1 \cap L_2 = L$ et $L_1 \cap L_2 = \emptyset$,
- $|S(L_1) - S(L_2)|$ minimale

Si on note $s = S(L)$, on a notamment $S(L_1) + S(L_2) = S$. On peut toujours, en permutant L_1 et L_2 supposer que $S(L_1) \leqslant S(L_2)$, ce qui donne $S(L_1) \leqslant \frac{s}{2} \leqslant L(S_2)$. On remarque alors que $|S(L_1) - S(L_2)|$ est minimale si et seulement si $\frac{s}{2} - S(L_1)$ est minimal. On ramène donc le problème à déterminer une sous-liste de L de sorte que $S(L') - \frac{s}{2}$ soit minimale (au dessus ou au dessous de 0, c'est pareil par symétrie)

- Une approche gloutonne ne fonctionne pas ici : on pourrait choisir le plus grand élément de L inférieur à $\frac{s}{2}$, puis lui ajouter le plus grand élément restant qui permet d'avoir une somme inférieure à $\frac{s}{2}$ et ainsi de suite. On trouve facilement des contre-exemples.
- Une approche dans laquelle on calcule toutes les sous-sommes possibles est exclue également dès que n est un peu grand (la complexité est un $O(n2^n)$ si $n = \operatorname{card} L$ - le 2^n pour le nombre de partitions et le n pour le calcul de la somme pour une partition donnée)

On aborde cela par une approche récursive (puis par programmation dynamique). L'idée est de se dire que `L[0]` peut faire parti de la parition minimale ou non :

- soit `L[0]` est dans la partition minimale et alors on doit chercher la sous-liste de `L[1:]` qui se rapproche le plus de $\frac{s}{2} - L[0]$,
- soit `L[0]` ni est pas et on doit chercher la sous-liste de `L[1:]` qui se rapproche le plus de $\frac{s}{2}$

Évidemment de cette manière, l'algorithme est de complexité exponentielle - on se rend compte qu'on passe beaucoup de temps à recalculer les mêmes sous-sommes... ce qui invite fortement à se diriger vers un algorithme par programmation dynamique. L'idée est de calculer les sommes qu'on obtient avec les sous-listes ne contenant que les k premiers termes : lorsqu'on ajoute le $k+1$ termes, les sommes atteignables sont les mêmes plus celles qu'on obtient en ajoutant le $k+1$-ème terme.

```
def sommes(L):
    """
    Entrée : liste d'entiers positifs L
    Sortie : (T,S) les sommes des partitions de L et la somme
             le plus proche de la moyenne
    """
    n = len(L)
    S = [0]*(n+1) # S[i] : dictionnaire avec les sous-sommes des i premiers termes
    S[0] = {0 : True}     # 0 est obtenu avec la somme vide
    for i in range(1,n+1): # on prend en compte le i-ème terme de la liste
        D = {}
        for val in S[i-1].keys():
            D[val] = True            # sommes déjà obtenues
            D[val+L[i-1]] = True     # celles avec le i-ème terme en plus
        S[i] = D                     # dictionnaire dont les clés sont les sous
                                     # sommes possibles avec i termes
    s = 0
    for x in L:                      # on calcule la somme totale
        s += x
    s = s//2                         # valeur moyenne
    while s not in S[n]:             # on descend dans la liste des valeurs à partir
        s -= 1                       # de la moyenne jusqu'à une somme obtenue
    return S,s
```

On peut tester sur une petite liste :

```
>>> sommes([2,4,5,3])
([{0: True},
  {0: True, 2: True},
 ...
  {0: True, 3: True, 5: True, 8: True, 4: True, 7: True, 9: True,
   12: True, 2: True, 10: True, 6: True, 11: True, 14: True}],
 7)
```

La complexité est à peu près quadratique (il faudrait être plus précis sur les temps de création et d'ajout dans les dictionnaires)

On peut reconstruire la partition à partir du tableau et de la valeur à atteindre. On peut tout faire en une seule fois :

```
def partition(L):
    """
    Entrée : liste d'entiers positifs L
    Sortie : (T,S) les sommes des partitions de L et la somme
             le plus proche de la moyenne
    """
    n = len(L)
    S = [0]*(n+1) # S[i] : dictionnaire avec les sous-sommes des i premiers termes
    S[0] = {0 : None}     # on met cette fois le dernier entier utilisé pour avoir
                          # cette somme
    for i in range(1,n+1):
        D = {}
        for val in S[i-1].keys():
            D[val] = S[i-1][val]
```

```
            if (val+L[i-1]) not in S[i-1]: # on ne considère que les nouvelles
                sommes
                D[val+L[i-1]] = L[i-1]      # pas obligatoire
        S[i] = D
    s = 0
    for x in L:
        s += x
    s = s//2
    while s not in S[n]:
        s -= 1

    v = s
    L2 = []
    for i in range(n,0,-1):
        if v>0 and v in S[i]:
            a = S[i][v]
            L2 = [a] + L2
            v = v-a

    return s,L2
```

ce qui donne

```
>>> partition([7,8,12,14,3,6,9,11,3])
(36, [7, 12, 14, 3])
>>> from random import randint
>>> L = [randint(0,1000) for i in range(100)]
>>> partition(L)
(25306, [617, 548, 377, 176, ..., 513, 272, 492, 936])
```

III.2 | Plus longue sous-séquence commune

Présentation du problème

On se donne deux chaînes de caractères C_1 et C_2. Une sous-séquence d'une chaîne de caractère C est une suite des caractères de la chaîne, pris dans l'ordre croissant mais pas forcément consécutif. Par exemple, si la chaîne est `"Informatique"`, une sous-séquence est `"Ifomaie"`. On cherche un algorithme pour détecter la plus longue sous-séquence commune entre deux chaînes de caractères (le résultat est commode avec les chaînes de caractères pour leur représentation. On peut évidemment le faire avec des listes/tableaux de tout type d'objets).

Expression récursive - sous-structure optimale

On se donne une chaîne `X`, de longueur n et on note $x_1,\dots,x_n$ ses caractères. De même on se donne une seconde chaîne Y de longueur m et $y_1,\dots,y_m$ ses caractères. On a alors deux situations :

- le dernier caractère est commun : $c = x_n = y_m$. Il est alors le dernier caractère de la plus longue sous-séquence commune et cette sous-séquence commune est obtenue en concaténant la plus longue sous-séquence commune de $x_1,\dots,x_{n-1}$ et $y_1,\dots,y_{m-1}$ et c.
- le dernier caractère est différent : l'un au moins de x_n et y_m n'est pas dans cette plus longue sous-séquence commune. On peut le retirer. La plus longue sous-séquence commune est la plus grande entre
  - → la plus longue sous-séquence commune entre $x_1,\dots,x_{n-1}$ et $y_1,\dots,y_m$,
  - → la plus longue sous-séquence commune entre $x_1,\dots,x_n$ et $y_1,\dots,y_{m-1}$,

On est donc ramené à chercher des sous-séquences maximales entre des sous-chaînes de X et Y : on a bien l'apparition de **sous-structure optimale** dans la solution. On trouve une illustration avec la figure 7.5

Une approche récursive directe va de nouveau amener à un problème de complexité exponentielle

Chevauchement des sous-problèmes

Le calcul de la plus longue sous-séquence commune d'une portions de chacune des deux chaînes est utilisé par beaucoup d'autres calculs de portions plus grande. On a donc bien un chevauchement très important des sous-problèmes, ce qui invite

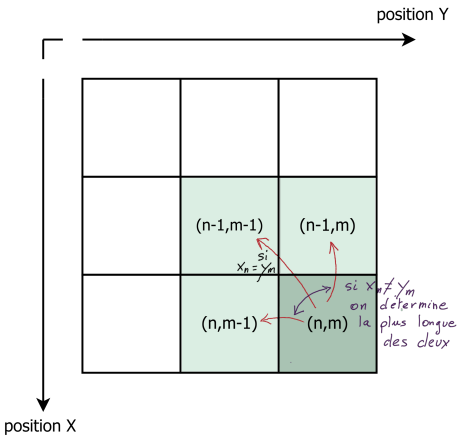


FIGURE 7.5 – Principe de recherche de la plus longue sous-séquence commune

de nouveau à un algorithme de programmation dynamique. Par exemple, dans la figure 7.6, on constate qu'il y a plusieurs chemins possibles qui passent par une même situation.

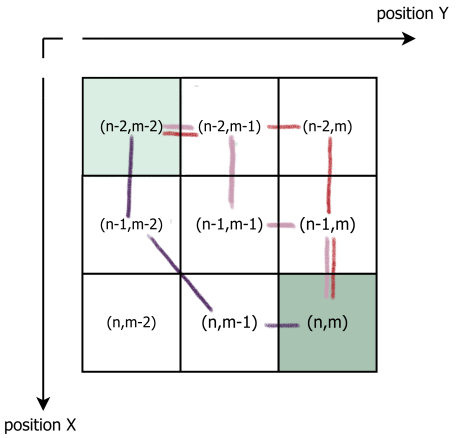


FIGURE 7.6 – Chevauchement des sous-problèmes - plusieurs chemins éventuels

Exercice 7.12 ***(Longueur - en descendant, avec mémoïsation)***

Écrire une version récursive de la recherche de la longueur de la plus longue sous-séquence commune en utilisant un tableau ou un dictionnaire pour mémoriser les calculs déjà fait (on ne demande que la longueur)

Exercice 7.13 ***(Longueur - de bas en haut)***

Écrire une version itérative qui remplit un tableau de taille $n \times m$, avec les longueurs des plus longues sous-séquences communes, en commençant par les cases simples, c'est-à-dire lorsque l'un des chaînes est de longueur nulle

Exercice 7.14 ***(Détermination de la plus longue sous-séquence commune)***

1. Ajuster le programme précédent pour qu'il crée en parallèle un second tableau avec un prédécesseur d'une case lorsqu'on la remplit (les cases de ce tableau peuvent contenir trois valeurs suivant que la case 'prédécesseur' est celle de gauche, du dessus ou en diagonale.
2. À partir de ce nouveau tableau, déterminer la plus longue sous-séquence commune
3. Écrire un programme `PLSC_chaine_final` qui renvoie la plus longue sous-séquence commune simplement à partir des deux chaînes (et qui ne calcule que le tableau des longueurs).

Chapitre 8 | Intelligence artificielle : apprentissage

Le but de ce chapitre est de présenter quelques algorithmes d'apprentissage. On dispose d'un certain nombres de données dans lesquelles on cherche à reconnaître certaines propriétés. Dans la suite, on prendra l'exemple de la reconnaissance de chiffres : on dispose d'une image d'un chiffre et on cherche à déterminer quel est ce chiffre. Il existe évidemment beaucoup d'autres domaines dans lesquels ces algorithmes peuvent s'appliquer : reconnaissance de formes, d'objets dans une image, algorithme de recommandation...

On va distinguer deux types d'apprentissage :

- **apprentissage supervisé :** on dispose de données de références (par exemple une banque d'images représentant des chiffres ainsi que le chiffre associé) qui vont servir à l'apprentissage. Une fois cette apprentissage réalisé, on se donne une nouvelle image et on cherche le chiffre associé. L'algorithme au programme est celui des *k-plus proches voisins* (souvent abrégé en KNN - K-Nearest Neighbors)
- **apprentissage non supervisé :** : cette fois on dispose de données mais sans information particulière sur ces données. On cherche alors à les regrouper en certaines catégories. L'algorithme au programme est celui des *k-moyennes* (ou M-means).

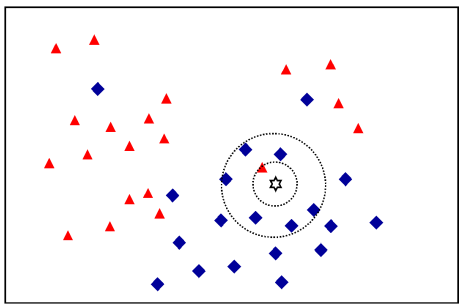
I | Algorithme des k-plus proches voisins avec distance euclidienne

I.1 | Principe et algorithme

On commence par l'apprentissage supervisé. L'idée principale est, pour un nouvel objet, de chercher quel est l'objet de référence le plus proche.

- on a donc besoin d'une notion de distance entre les objets. Ici, on se contentera de voir les objets comme un ensemble de points de $\mathbb{R}^n$ et de considérer la distance euclidienne. Une image de taille $n \times n$ en niveau de gris (de 0 à 255 par exemple) peut être vue comme un vecteur à n^2 composantes où les composantes sont les valeurs - de 0 à 255, ou entre 0 et 1 en renormalisant - des pixels.
- s'intéresser à l'objet le plus proche est souvent trop restrictif. On va plutôt s'intéresser à savoir quel est l'objet de référence qui apparaît le plus dans un voisinage du nouveau point.

Dans l'exemple suivant, on cherche à catégoriser un nouvel objet - l'étoile. On dispose de deux types de données connues : celles qui sont des triangles rouges et celles qui sont des carrés bleus.



L'objet le plus proche est un triangle rouge mais essentiellement les objets autour sont des carrés bleus. Si on cherche le nombre d'objets d'un certain type autour de l'étoile, on a rapidement plus de carrés que de triangle. On cherche quels sont les k-plus proches voisins de l'étoile (k à voir) et parmi ses objets on regarde le caractère dominant. Si on prend les 5 plus proches objets de l'étoile, on aura un triangle et quatre carrés - on a donc tendance à catégoriser ce nouvel objet (l'étoile) comme un carré bleu.

Algorithme KNN

Données : `L` données en liste (coordonnées, catégorie) , `x` nouveau point, `k` un entier
Sorties : la catégorie de `x` obtenu par l'algorithme KNN
début
    Calculer les distances entre `x` et les points de `L`
    Trier la liste des distances et conserver les indices des `k` plus proches
    Parmi ces k plus proches, déterminer la catégorie dominante `C` **retourner** C

Il y a cependant plusieurs points à évoquer :

- calculer la liste des distances et la trier est le but, mais il faut aussi conserver les points correspondants à ces distances - ou à défaut, conserver la catégorie correspondante. On peut créer des listes de couples `[d,c]` où `d` est la distance entre les deux objets et `c` la catégorie de l'objet correspondant et trier suivant la première valeur. C'est ce que fait la fonction de tri `sorted` de Python. Sa complexité est quasi-linéaire.
- puisqu'on sait qu'on veut garder les k plus petites valeurs, on peut également reprogrammer un algorithme qui s'inspire du tri par insertion (chercher les k plus petites valeurs d'une liste) - sa complexité sera en $O(k.n)$ où n est le nombre de données.
- afin daccélérer les calculs, il serait profitable d'avoir un moyen d'éliminer rapidement les points qui seraient trop loin du nouveau point (mais on n'en discutera pas ici).

Remarque : la fonction `sorted` peut s'utiliser en précisant quelle fonction est utilisée pour comparer des éléments, ou quelle valeur est utilisée pour le tri. Par exemple, si on dispose d'une liste d'indices `L` et une liste des distances associées `D` (donne la distance entre l'objet et celui de l'indice), la syntaxe

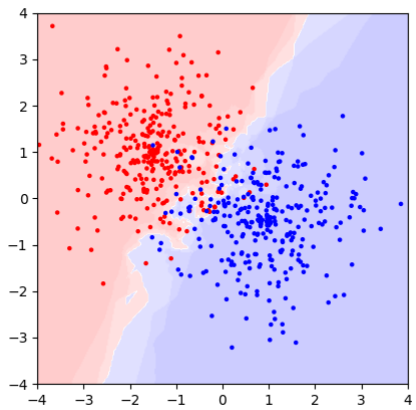
```
>>> sorted(L,key = lambda j:D[j])
```

permet de trier la liste d'indices `L` suivant les valeurs des D - ce sont bien les indices qui sont triés.

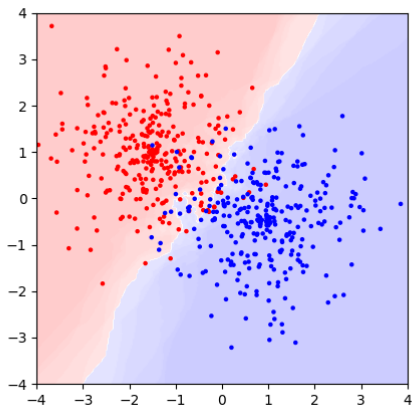
I.2 | Exemples (à voir en couleur)

On dispose de deux catégories (rouge et bleu) et on représente les zones détectées par l'algorithme KNN pour différentes valeurs de k. On fait apparaître différentes zones. Par exemple, pour $k = 3$, on a celles du type $(3,0)$ (3 rouges et 0 bleu parmi les 3 plus proches), les $(2,1)$, $(1,2)$ et $(0,3)$.

Différentes zones - cas d'une séparation linéaire

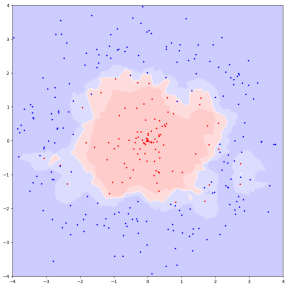


$k = 3$

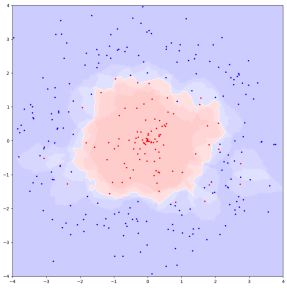


$k = 11$

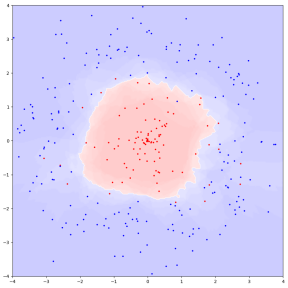
Différentes zones - cas d'une séparation circulaire



$k = 3$



$k = 5$



$k = 11$

I.3 | Choix de k, choix des données, matrice de confusion

Comment choisir k ? Prendre une valeur trop petite risque de conduire à des cas mal déterminés notamment à cause de parasites (repensez au cas de $k = 1$). Prendre k trop grand risque de ne pas améliorer la détection.

Souvent le nouveau point risque d'être proche de deux catégories. Il est assez judicieux de choisir un entier impair pour k afin d'avoir une catégorie dominante (évidemment cela peut parfois poser problème, par exemple si il y a trois catégories proches et une réponse de type $(2,2,1)$ pour les voisins)

Afin de tester si les données d'entrée et le choix de k sont raisonnables, on peut utiliser l'algorithme sur d'autres données connues et voir le comportement de l'algorithme. En pratique, on sépare les données d'apprentissage en deux jeux de données :

- les données d'apprentissage : ce sont celles qui vont servir pour appliquer l'algorithme,
- les données de tests : on connaît leur catégorie, on applique l'algorithme sur ces données de tests avec les données d'apprentissage (et éventuellement différentes valeurs de k)

On peut alors regarder la quantité de valeurs bien détectées par l'algorithme (et si elle ne convient pas, changer les données d'apprentissage et/ou la valeur de k).

Pour présenter cela, on peut créer *la matrice de confusion* associée à ces données : c'est un tableau à deux entrées dans lequel les lignes représentent les valeurs détectées par l'algorithme, les colonnes représentent les catégories réelles des objets. On remplit ce tableau avec le jeu de données de tests (on peut mettre soit le nombre de valeurs détectées - par exemple en ligne i, colonne j, on aura le nombre d'objets du jeu de tests qui sont de la catégorie j et qui ont été détectés comme appartenant à la catégorie i par l'algorithme. Le choix est « bon », lorsque les valeurs sur la diagonales sont grandes et les autres petites.

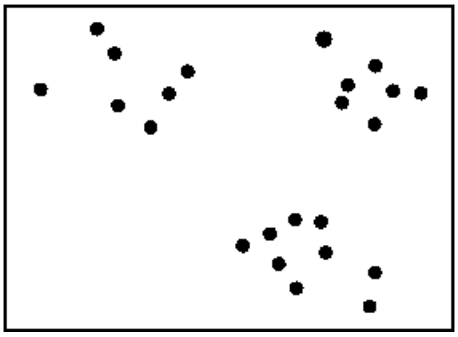
Remarque : : à la place de quantités, on peut également mettre le pourcentage par rapport à la catégorie correcte (on divise chaque case par la somme des valeurs sur la colonne correspondante).

II | Algorithme des k-moyennes

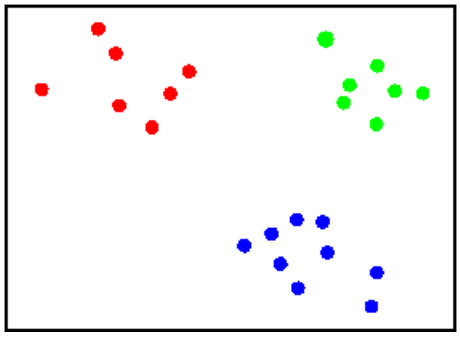
Dans cette partie, on présente un algorithme d'apprentissage non supervisé : on dispose d'un jeu de données qu'on cherche à regrouper par catégories, sans avoir de catégories de référence (ni savoir combien de catégories on doit obtenir). De nouveau, on se place dans le cadre d'un jeu de données qui sont des points de $\mathbb{R}^n$ muni de la distance euclidienne. On cherche alors à regrouper les données par paquets d'objets proches entre eux. Par exemple, on a envie de regrouper les données brutes suivantes en trois paquets :

Le principe de l'algorithme est simple :

Regroupement des données



Données brutes



Données regroupées

Algorithme des k-moyennes

Données : L données en liste de points de $\mathbb{R}^n$, un entier k
Sorties : Un ensemble de classes de points
début
  Fixer aléatoirement k centres $c_1,\dots,c_k$ (par exemple choisir k données)
  tant que *la situation « évolue »* **faire**
    Associer chacune des données au centre le plus proche pour créer des classes $C_1,\dots,C_k$
    Calculer les barycentres $c_1,\dots,c_k$ des points des classes $C_1,\dots,C_k$
  retourner $C_1,\dots,C_k$

De nouveau, cela reste assez empirique (on peut malgré tout montrer que l'algorithme converge c'est-à-dire qu'à partir d'un moment les classes n'évoluent plus). De même la valeur de k est un paramètre de l'algorithme.

On pourrait donner un critère pour évaluer la situation. On peut, par exemple, calculer la somme des distances des points au barycentre de leur classe et essayer faire baisser cette quantité :

- en changeant les centres de départ
- en changeant la valeur de k

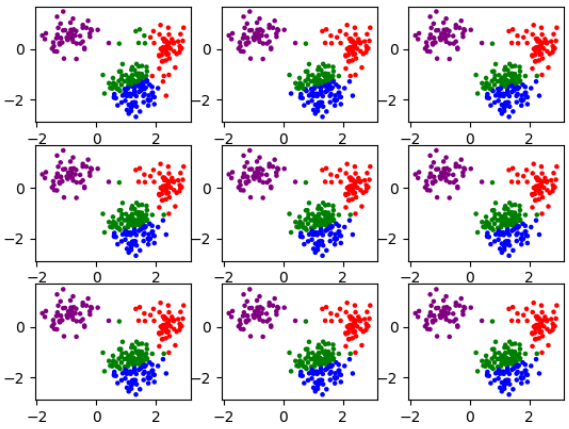
On peut également envisager d'autres manières de répartir au départ, par exemple affecter chaque données à une catégorie aléatoire, calculer les barycentres et appliquer l'algorithme.

Dans la suite, on trouve quelques exemples (à voir en couleur) obtenus avec différentes étapes de l'algorithme.

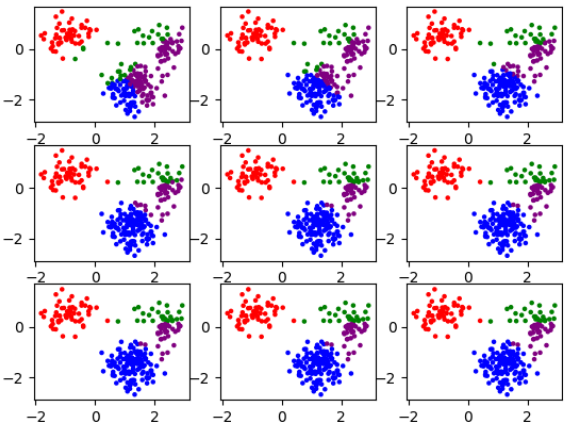
On remarque que l'algorithme peut converger vers des situations différentes en fonction du choix aléatoire des centres de départ. Le choix d'une catégorie au hasard n'apporte finalement rien, le calcul des premiers barycentres revient plus ou moins à choisir ces centres au hasard (si on a suffisamment de points) - l'intérêt est donc nul.

Remarque : ces deux algorithmes peuvent être combinés. On dispose d'un jeu de données sans information. On commence par réaliser une catégorisation avec l'algorithme des k-moyennes. Une fois cela réalisé, cela permet d'appliquer l'algorithme des k-plus proches voisins lorsqu'on rencontre de nouvelles données qu'on cherche à identifier.

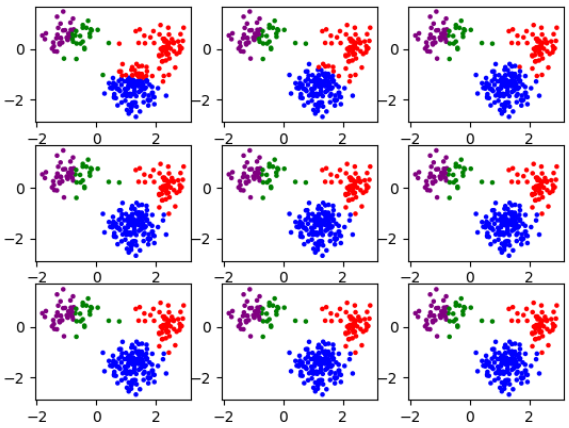
Application de l'algorithme des k-moyennes



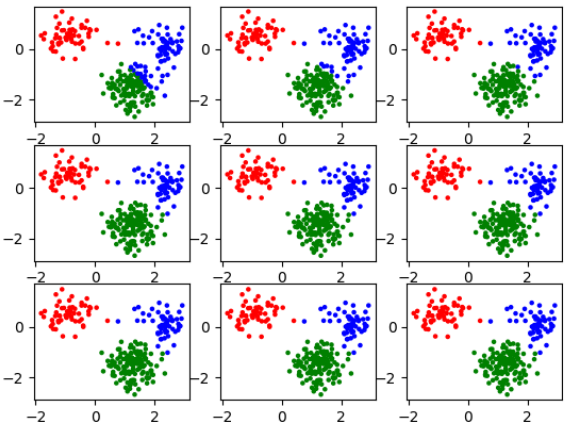
Test 1 - $k = 4$, centre au hasard



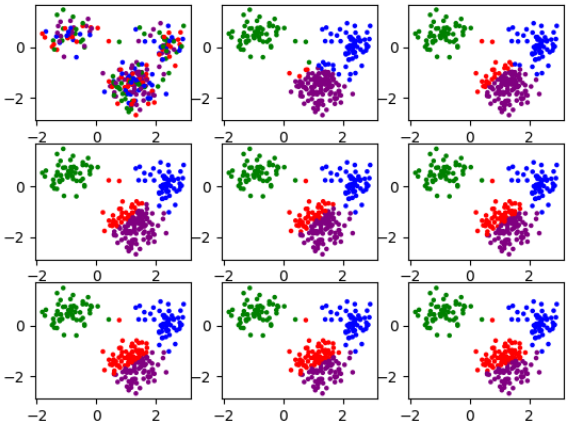
Test 2 - $k = 4$, centre au hasard



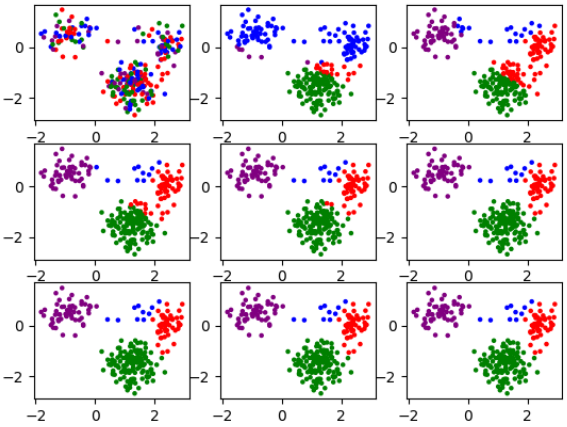
Test 3 - $k = 4$, centre au hasard



Test 4 - $k = 3$, centre au hasard



Test 5 - $k = 4$, catégorie au hasard



Test 6 - $k = 4$, catégorie au hasard

Chapitre 9 | Théorie des jeux

I | Présentation

On s'intéresse dans cette partie à un cas particuliers de la théorie des jeux, à savoir certains jeux à 2 joueurs où ils jouent à tour de rôle. On suppose de plus qu'on a ces contraintes :

- à chaque étape, un joueur doit choisir parmi un nombre fini de possibilités,
- le jeu est à information complète, c'est-à-dire qu'elles sont toutes visibles - cela exclut par exemple les jeux de cartes avec des cartes cachées,
- le jeu est sans mémoire : le choix à un moment ne dépend pas des choix précédents mais seulement de la situation actuelle du jeu (cela exclut le jeu d'échec à cause de la possibilité d'un roque unique)
- le jeu est impartial : le choix à un moment ne dépend pas du joueur
- le jeu est sans hasard

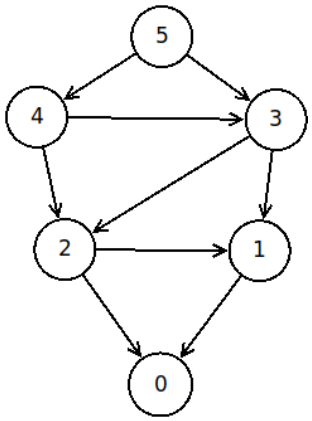
I.1 | Exemples

Voici quelques exemples de jeux qu'on peut étudier :

- le morpion (tic-tac-toe) : un grille de $3x3$ dans laquelle les joueurs mettent successivement une croix ou un rond - le premier qui en aligne 3 a gagné
- le jeu des allumettes (cas particulier d'un jeu de Nim) : on a un tas de N allumettes et à tour de rôle un joueur peut en retirer entre une et k. Celui qui prend la dernière allumette gagne.
- le jeu de Marienbad : on a quatre tas constitués respectivement de $1,3,5$ et 7 allumettes. À son tour un joueur peut retirer autant d'allumettes qu'il souhaite mais d'un seul tas. Celui qui prend la dernière perd
- plus généralement, les jeux de Nim : un mélange des deux précédents - on a plusieurs tas d'objets et on peut retirer un nombre maximum d'objets dans un tas (ou un nombre quelconque). Le gagnant est celui qui prend le dernier objet - ou bien c'est le perdant...
- le jeu de « Chomp » : des carrés de chocolat sont disposés sur une grille rectangulaire de taille $p \times q$. À tour de rôle , un joueur choisit un carré et prend tous les carrés restants dans la zone rectangulaire ayant ce carré pour coin inférieur gauche. Celui qui prend le dernier carré a perdu.

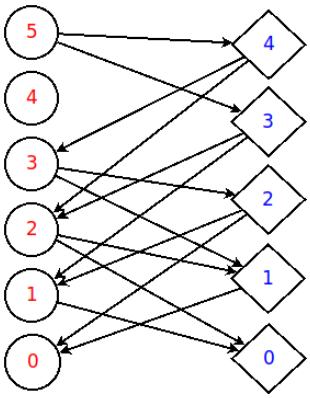
II | Modélisation

On modélise ces jeux par un graphe dont les sommets sont les situations possibles et les arcs représentent les coups jouables à partir d'un sommet. L'exemple suivant illustre le jeu de Nim avec 5 allumettes. À chaque tour, on doit retirer 1 ou 2 allumettes.



Il est beaucoup plus pratique de changer ce graphe sous une autre forme : les sommets accessibles au premier joueur J1 et ceux accessibles au second J2. Le jeu va être modélisé par un graphe $G = (S, A)$ biparti : on peut séparer les sommets en deux

sous-ensembles disjoints $S = S_1 \uplus S_2$ où S_1 sont les sommets accessibles à J1 et ceux accessibles à J2, avec comme condition que si $a = (s, s')$ est un arc alors les sommets s et s' sont chacun dans l'un des ensembles S_1 et S_2 (en clair, chaque arc fait passer d'un ensemble de sommets à l'autre). On reprend l'exemple précédent (en pratique, il suffit de transformer les sommets s en couple $(1, s)$ et $(2, s)$ afin de désigner le joueur devant effectuer son coup) :



On évidemment retirer le sommet isolé (4 allumettes pour le premier joueur).
Un peu de vocabulaire sur *les jeux d'accessibilité* :

- graphe du jeu ou *arène* : un graphe (G, S) biparti avec $S = S_1 \uplus S_2$ où S_1 et S_2 sont les sommets accessibles respectivement pour J1 et J2
- *une situation* : un sommet du graphe
- *décision/choix/coup* : un arc du graphe
- chaque joueur possède un *objectif/but*, c'est-à-dire un sommet (ou un ensemble de sommets) qu'il cherche à atteindre.
- si le joueur 1 gagne, le joueur 2 perd (et réciproquement) - le jeu est à somme nulle. On peut également avoir des situations de match nul.
- les *états finals/finaux* sont de 3 types : ceux gagnants pour J1, ceux gagnants pour J2 et ceux de math nul. Aucun arc ne part de l'un de ces sommets.
- *une partie* est un chemin fini de sommets (en général qui débute avec une situation initiale s_0)

III | Stratégies, stratégies gagnantes

III.1 | Vocabulaire

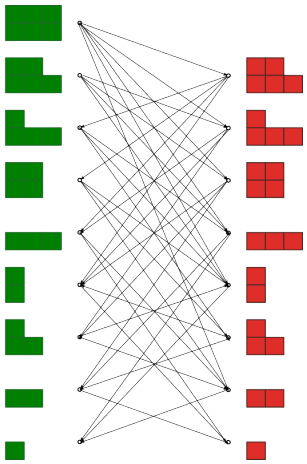
Encore un peu de vocabulaire :

- *une stratégie* est une fonction $\varphi : S \to S$ telle que si $s \in S$ et $s' = \varphi(s)$, alors (s, s') est un arc du graphe (en clair, une stratégie est une fonction qui indique quel coup effectuer à partir de n'importe quel coup du graphe)
- une stratégie φ est *gagnante* à partir d'un sommet s_1 pour un joueur lorsque, quels que soient les coups de l'autre joueur, le joueur qui suit la stratégie φ l'emporte
- une position s est gagnante pour un joueur lorsqu'il existe une stratégie φ qui est une stratégie gagnante à partir de cette situation.

III.2 | Détermination des stratégies et positions gagnantes

Pour comprendre comment déterminer ces éléments, on considère le jeu de chomp sur un rectangle $(2,3)$. Rappelons les règles : un joueur choisit un carré de chocolat (ou autre) et mange tous les carrés situés dans le rectangle à droite et au dessus (le carré choisit forme le coin). On continue jusqu'à ce qu'un joueur mange le carré en bas à gauche (empoissonné, périmé, au lait... au choix)

L'exemple qui suit correspond au graphe du jeu de Chomp avec 2×3 carrés au départ.



Le premier joueur est le joueur qui a ses sommets à gauche (en vert). Il joue pour aller sur le côté droit (rouge). C'est ensuite le second joueur qui retire ses carrés.

Le but est de trouver une stratégie qui obligera le joueur J2 à prendre le dernier carré. Pour cela on remarque que le premier joueur a trois situations qui peuvent amener à la victoire. Il va donc essayer d'arriver à ces positions gagnantes. Le problème est que J2 réfléchit et donc veut les éviter... il faut donc l'amener dans une situation où quel que soit son coup, il tombera dans une position gagnante pour J1...

n voit donc apparaître une méthode pour « remonter » les positions gagnantes - avec une disymétrie entre les deux joueurs : pour le premier il suffit d'être dans une situation qui l'amènera à la victoire alors que pour le joueur J2, il faut l'amener dans une situation où tous ses coups le conduiront à la défaite.

Plus précisément, regardons les différentes étapes en appliquant cette méthode. On part de la situation gagnante pour J1 (carré seul à droite) et on remonte étape par étape tant qu'on peut :

De façon plus théorique, on va calculer les attracteurs d'une partie F des sommets du graphe $G = (S, A)$, c'est-à-dire l'ensemble des positions gagnantes (si gagner signifie arriver dans F). On définit cela par récurrence en distinguant le joueur 1 qui peut choisir ses coups et le joueur 2 qu'on doit obliger à perdre. On note de nouveau $S = S_1 \uplus S_2$ où S_1 et S_2 sont les sommets accessibles respectivement à J1 et J2.

- on part de $\mathcal{A}_0(F) = F$
- on définit alors par récurrence : pour $i \in \mathbb{N}$,
  - → $\mathcal{A}'_i(F) = \mathcal{A}_i(F) \cup \{s \in S_1 / \exists (s, s') \in A, s' \in \mathcal{A}_i(F)\}$ (on ajoute les sommets du joueur 1 qui peuvent aboutir à une situation gagnante
  - → $\mathcal{A}_{i+1}(F) = \mathcal{A}'_i(F) \cup \{s \in S_2 / \forall (s, s') \in A, s' \in \mathcal{A}'_i(F)\}$ (on ajoute les sommets du joueur 2 pour lesquels tous les coups donnent des situations gagnantes)

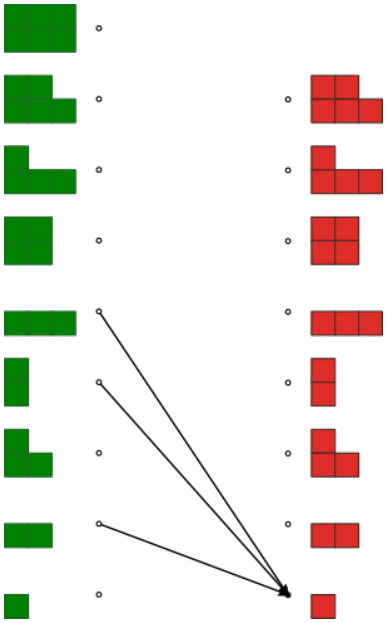
Puisque la suite $\mathcal{A}_i(F)$ est croissante (et dans un ensemble fini), elle est stationnaire.

Remarque : on peut également définir la suite $\mathcal{A}_i(F)$ en une seule étape avec un décalage sur les sommets de S_2 :

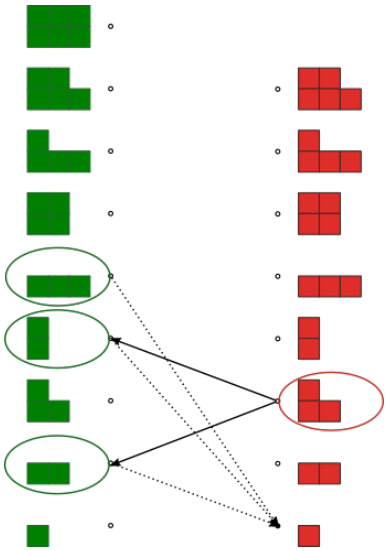
$$\mathcal{A}_{i+1}(F) = \mathcal{A}_i(F) \cup \{s \in S_1 / \exists (s, s') \in A, s' \in \mathcal{A}_i(F)\} \cup \{s \in S_2 / \forall (s, s') \in A, s' \in \mathcal{A}_i(F)\}$$

Remarque : la construction précédente fait apparaître un parcours des sommets de S_1 ou S_2. Afin de ne pas parcourir tous les sommets $s \in S_1$ et regarder s'il existe un arc d'origine s qui tombe dans $\mathcal{A}_i(F)$, on peut à la place commencer par créer un dictionnaire des prédécesseurs pour chaque sommet de S. On parcourt alors les sommets de $\mathcal{A}_i(F) \cap S_2$ et on ajoute les prédécesseurs qui ne sont pas déjà dans $\mathcal{A}_i(F)$.

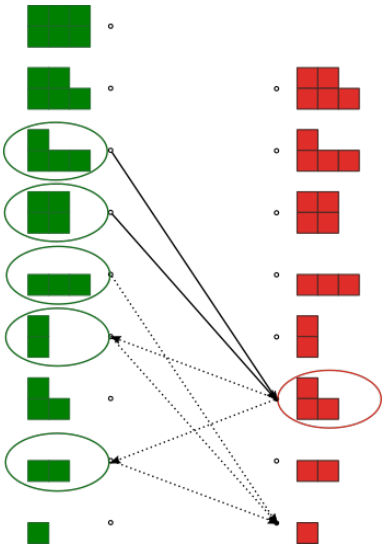
Illustration du calcul des positions gagnantes dans le jeu de chomp



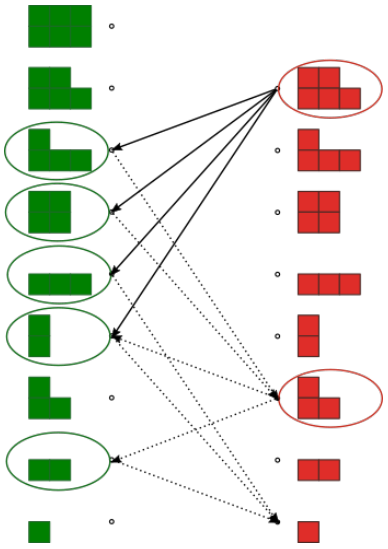
3 situations qui donnent la victoire au premier joueur



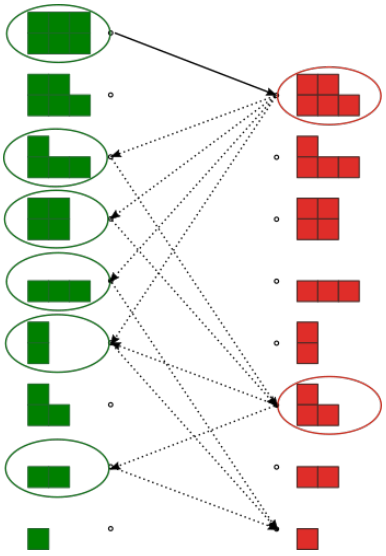
Une position forcément perdante pour J2 → position gagnante pour J1



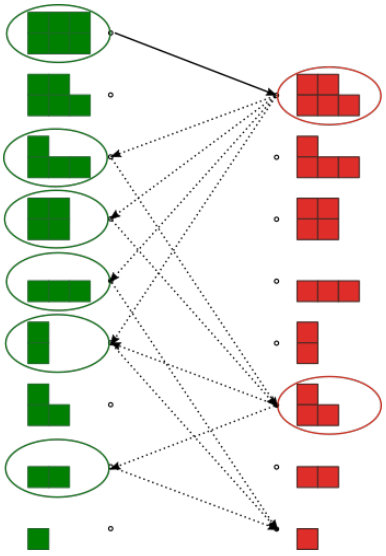
De nouvelles situations qui aboutissent à la victoire



Cette situation tombe uniquement sur des positions gagnantes



La situation de départ permet d'arriver à cette situation



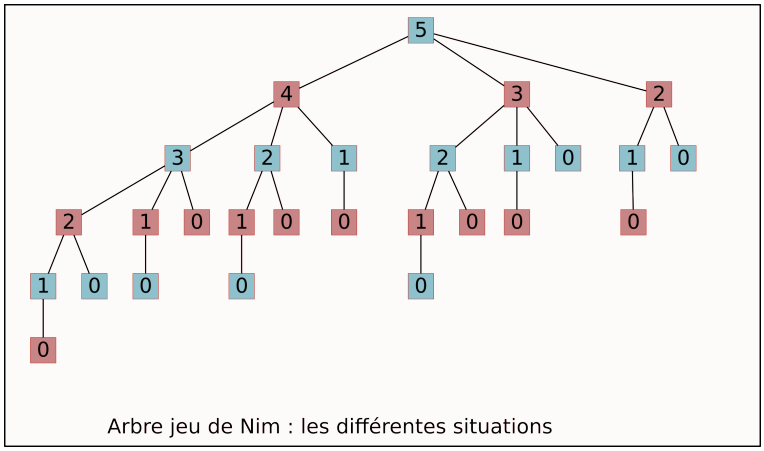
On a l'ensemble des situations gagnantes et la stratégie pour y arriver

IV | Heuristique et algorithme min-max

L'un des problèmes rencontré dans la méthode précédente et que le graphe du jeux est très très gros. Par exemple, considérons le jeu du « puissance 4 » où le vainqueur doit aligner 4 pions de sa couleur. La grille classique est de taille $6 \times 7 = 42$ et dans la plupart des situations, il y a 8 coups possibles... le graphe associé au jeu est énorme!!! De plus le nombre de situations gagnantes est également très grand. Il semble donc extrêmement difficile d'imaginer qu'on va réussir à calculer les positions/stratégies gagnantes pour ce jeux. On doit donc proposer d'autres méthodes.

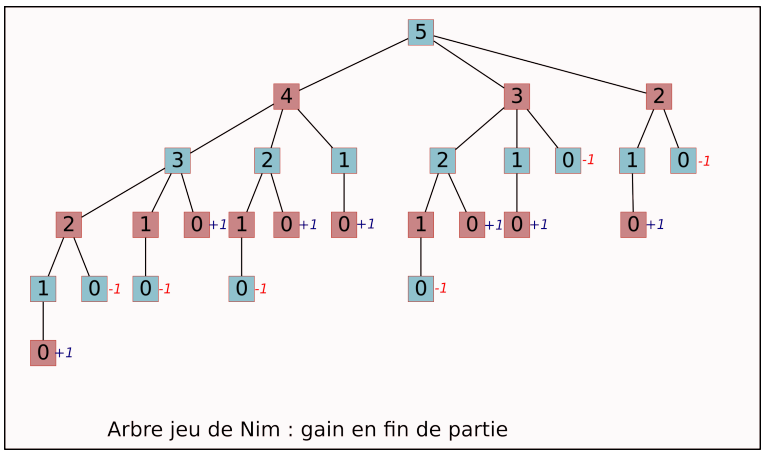
IV.1 | Exemple simple : jeu de Nim

Reprenons l'exemple du jeu de Nim avec 5 allumettes au départ et la possibilité d'en retirer de une à trois. On représente un arbre décrivant les déroulements complets du jeu (pour simplifier, un arbre sera un graphe particulier, sans cycle, avec un sommet désigné comme racine) :



Arbre jeu de Nim : les différentes situations

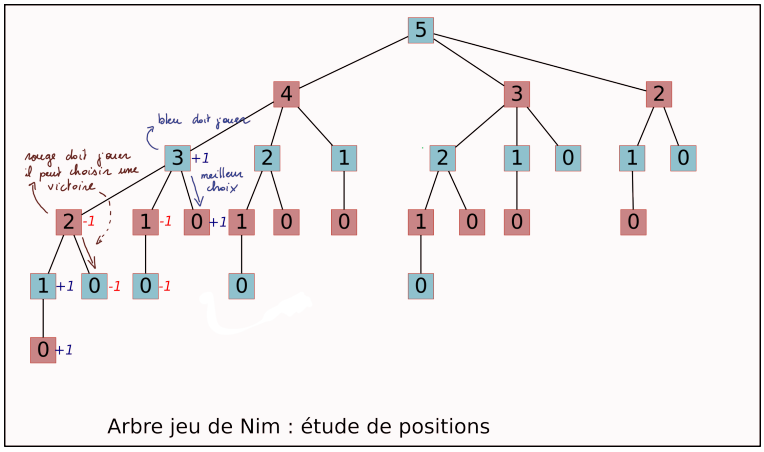
On a une alternance entre les tours du premier joueur et les tours du second. Les positions finales gagnantes (toujours pour J1) sont les sommets avec une étiquette 0 lorsque c'est le tour du joueur 2, alors que les positions perdantes sont celles où le sommet est étiqueté 0 et lorsque c'est au tour du joueur J1. On affecte $+1$ pour ces sommets gagnants et -1 pour les perdants.



Arbre jeu de Nim : gain en fin de partie

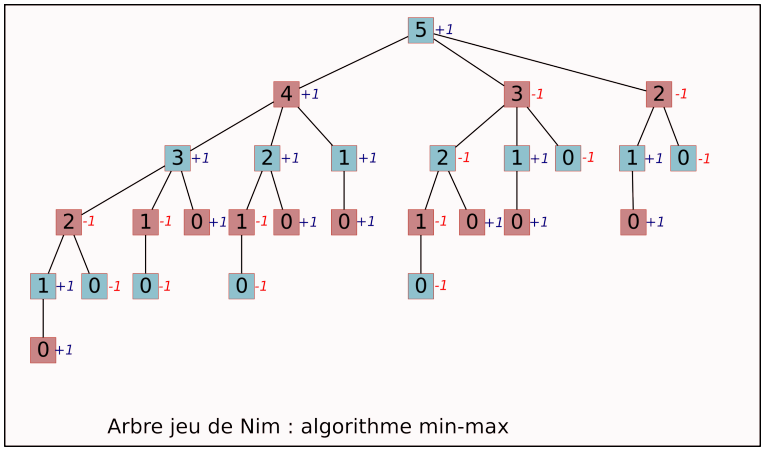
On va alors remonter ces positions gagantes/perdantes avec comme principe que chaque joueur joue au mieux. Lorsque l'un des coups de J1 permet d'aboutir à une situation gagante, ce sommet reprend la valeur $+1$. Au contraire, lorsque c'est à J2 de jouer, si l'un de ses coups lui permet d'être certain de gagner, il va le réaliser. On va donc remonter les valeurs $+1$ et -1 avec le principe suivant :

- lorsque c'est à $J1$ de jouer, il choisit le meilleur coup pour lui, c'est-à-dire celui qui maximise la valeur sur les coups suivants
- lorsque c'est à $J2$ de jouer, il choisit le meilleur coup pour lui, c'est-à-dire celui qui minimise la valeur sur les coups suivants (on prend toujours le point de vue du gain pour le joueur 1).



Arbre jeu de Nim : étude de positions

Ce qui donne après avoir tout remonté :



Arbre jeu de Nim : algorithme min-max

Ainsi la situation de départ est gagnante pour J1. Son premier coup doit donc être celui où il retire une allumette. Quel que soit le coup de son adversaire, il sait comment jouer pour gagner.

Remarque : évidemment on connaît une stratégie pour gagner... il suffit qu'après son coup, il reste un nombre d'allumettes multiple de 4, ainsi à chaque tour il pourra jouer le complémentaire à 4 de son adversaire, jusqu'à prendre la dernière allumette. Si le nombre d'allumettes de départs n'est pas un multiple de 4, il est certain de gagner. En revanche si ce nombre est multiple de 4, c'est le joueur J2 qui a une stratégie gagnante.

IV.2 | Généralisation et heuristique

L'algorithme du min-max est donc assez simple. Il est récursif. On affecte un gain aux sommets qui n'ont plus de successeurs (dans le jeu de Nim, on a affecté ± 1). Pour un sommet donné, je gain (pour le joueur 1) est

- le maximum des gains des différents sommets successeurs si c'est au tour de J1 de jouer,
- le minimum des gains des différents sommets successeurs si c'est au tour de J2 de jouer.

On est de nouveau confronté au problème de la taille du graphe. Si on reprend le jeu de puissance 4, on peut avoir jusqu'à 42 coups (cela donne la hauteur du graphe) et chaque étage est approximativement 8 fois plus grand que le précédente (c'est très approximatif...). On ne peut donc pas explorer récursivement tout le graphe jusqu'à arriver à l'ensemble des positions finales. On va donc se limiter à une certaine profondeur.

Nouveau problème : si on s'arrête une certaine profondeur, on ne sait pas quel est le gain des sommets à cette profondeur puisqu'on n'est pas allé jusqu'au bout. On doit évaluer l'intérêt d'un certain sommet en fonction de sa situation. C'est à ce moment qu'on introduit le notion d'*heuristique* : une fonction qui va donner un score à un sommet donné. On pourra alors appliquer l'algorithme min-max en s'arrêtant au bout de n étapes (n étant à déterminer en fonction de la puissance de l'ordinateur / du niveau de difficulté de l'intelligence artificielle qu'on met en place). Par exemple pour le puissance 4, on situation peut obtenir un score en fonction des pions déjà joué : on met plus de points pour les pions plutôt au centre, et moins à ceux sur les bords ; on peut compter le nombre d'alignements de 2, de 3 jetons (avec un certain poids)... tout cela pour dire qu'on est plus ou moins proche d'une situation gagnante.

Algorithme du min-max avec heuristique

Données : `G` un arbre de jeux, s la position actuelle, n la profondeur de recherche, j le joueur, h une fonction heuristique qui évalue un sommet
Sorties : le gain maximal qu'on peut obtenir / le meilleur coup à jouer
début
  si $n = 0$ **alors**
    retourner $h(s)$
  sinon
    si `joueur(s)=j` **alors**
      retourner *le maximum des* `minmax(G,s',j,n-1,h)` *où* `s'` *décrit les successeurs de* `s`
    sinon
      retourner *le minimum des* `minmax(G,s',j,n-1,h)` *où* `s'` *décrit les successeurs de* `s`

Chapitre 10 | CCINP MP

I | CCINP MP 2024

Pour $n \in \mathbb{N}^*$, on note S_n le groupe des permutations de l'ensemble $[\![0; n-1]\!]$. Une permutation de S_n sera représentée en Python par une liste, dont l'élément d'indice `i` est l'image de `i` par cette permutation. Par exemple, la liste $[3,1,0,2]$ représente la permutation $\sigma \in S_4$ définie par $\sigma(0) = 3$, $\sigma(1) = 1$, $\sigma(2) = 0$ et $\sigma(3) = 2$.

Dans tout l'exercice, on pourra utiliser librement les tests Python du type `x in L` (respectivement `x not in L`) permettant de vérifier si `x` est présent dans la liste `L` (respectivement de vérifier si `x` n'est pas présent dans la liste `L`).

1. Si `s` est une liste Python représentant une permutation de S_4, quelle instruction Python permet de trouver l'image de `1` par cette permutation ? Quelle liste Python représente la transposition $(2\ 3) \in S_4$?
2. Écrire une fonction Python `comp(s1,s2)` prenant en entrée deux listes représentant des permutations σ_1 et σ_2 du même groupe de permutations et renvoyant la liste représentant la permutation $\sigma_1 \circ \sigma_2$.
3. Écrire une fonction Python `inv(s)` prenant en entrée une liste représentant une permutation σ et renvoyant la liste représentant σ^{-1}.
4. On souhaite tester si un sous-ensemble G de S_n est ou non un sous-groupe de S_n. Écrire une fonction Python `groupe(G)` prenant en entrée une liste de listes, où chaque sous-liste représente une permutation de S_n et renvoyant `True` s'il s'agit bien d'un sous-groupe de S_n, `False` sinon.
5. Écrire une fonction Python `cyclique(s)` prenant en entrée une liste `s` représentant une permutation σ de S_n et renvoyant le sous-groupe de S_n engendré par σ sous la forme d'une liste de listes.

II | CCINP MP 2023

Dans cet exercice d'informatique commune, on se propose d'écrire des algorithmes dans le but de faire du calcul matriciel et plus particulièrement afin d'utiliser les matrices d'adjacence d'un graphe. Les algorithmes demandés doivent être écrits en langage **Python**. On sera très attentif à la rédaction et notamment à l'indentation du code. L'usage de toute librairie est **interdit**.

Notation

Les matrices sont carrées et représentées par des listes dont les éléments correspondent aux lignes de la matrice. Par exemple, la matrice $A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ est représentée par la liste `[[1,2],[3,4]]`.

Dans la suite, pour définir la matrice d'adjacence $A \in \mathcal{M}_n(\mathbb{R})$ d'un graphe ayant n sommets, on numérote ses sommets de 0 à $n-1$.

1. Écrire une fonction `produit(A, B)` prenant en arguments deux matrices carrées A et B de mêmes dimensions et qui renvoie $A \times B$ le produit de la matrice A par la matrice B.
2. Écrire une fonction `oriente(A)` prenant en argument la matrice d'adjacence A d'un graphe et qui retourne `True` si le graphe est orienté et `False` sinon.
3. On admet que le nombre de chemins de longueur p reliant i et j dans un graphe de matrice d'adjacence A est égal au coefficient d'indice (i, j) de la matrice A^p.

   Écrire une fonction `distance(A,i,j)` où A est la matrice d'adjacence d'un graphe et qui renvoie le nombre minimal d'arêtes que l'on doit parcourir pour atteindre le sommet j depuis le sommet i (on suppose qu'un tel chemin existe).

On considère deux tables : CLIENTS e PARTENAIRES. La première contient des informations sur les clients et la deuxième permet d'identifier qui sont les partenaires des clients.

La table CLIENTS contient les attributs suivants :

- `id` : identifiant d'un individu (entier), clé primaire ;
- `nom` (chaîne de caractères) ;
- `prenom` (chaîne de caractères) ;
- `ville` (chaîne de caractères) ;
- `email` (chaîne de caractères).

La table PARTENAIRES contient les attributs suivants :

- `id` : identifiant de suivi (entier), clé primaire :
- `id_client` : identifiant du client représenté par l'attribut id dans la table CLIENTS (entier) ;
- `partenaire` : nom du partenaire (chaîne de caractères).

4. Écrire une requête SQL permettant d'extraire les identifiants de tous les clients provenant de la ville de "Toulouse".
5. Écrire une requête SQL permettant d'extraire les emails de tous les clients ayant "SCEI" comme partenaire.

III | CCINP MP 2022

Quelques questions intercalées

IV | CCINP MP 2021

Partie I – Algorithmique : calcul de zêta aux entiers pairs

La suite des nombres de Bernoulli notée $(b_n)_{n\in\mathbb{N}}$ est définie par :

$$b_0 = 1, \qquad \forall n \geqslant 1, b_n = \frac{-1}{n+1}\sum_{k=0}^{n-1}\binom{k}{n+1}b_k.$$

Leonhard Euler (1707-1783) a démontré la formule suivante qui exprime les nombres $\zeta(2k)$ à l'aide des nombres de Bernoulli :

$$\forall k \in \mathbb{N}^*, \zeta(2k) = \frac{(-1)^{k-1}2^{2k-1}\pi^{2k}b_{2k}}{(2k)!}.$$

Dans cette partie (informatique pour tous), on se propose de programmer le calcul des nombres de Bernoulli b_n afin d'obtenir des valeurs exactes de $\zeta(2k)$.

Les algorithmes demandés doivent être écrits en langage Python. On sera très attentif à la rédaction du code notamment à l'indentation.

1. Écrire une fonction `factorielle(n)` qui renvoie la factorielle d'un entier $n \in \mathbb{N}$.
2. On considère la fonction Python suivante `binom(n, p)` qui renvoie le coefficient binomial $\binom{p}{n}$:

```
def binom(n, p):
    if not (0 <= p <= n):
        return 0
    return factorielle(n) // (factorielle(p) * factorielle(n - p))
```

Combien de multiplications sont effectuées lorsque l'on exécute `binom(30, 10)` ?
Expliquer pourquoi il est possible de réduire ce nombre de multiplications à 20 ? Quel serait le type du résultat renvoyé si l'on remplaçait la dernière ligne de la fonction `binom` par

```
return factorielle(n) / (factorielle(p) * factorielle(n - p))
```

3. Démontrer que, pour $n \geqslant p \geqslant 1$, on a

$$\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}.$$

En déduire une fonction récursive `binom_rec(n, p)` qui renvoie le coefficient binomial $\binom{n}{p}$.

4. Écrire une fonction non récursive `bernoulli(n)` qui renvoie une valeur approchée du nombre rationnel b_n. On pourra utiliser librement une fonction `binomial(n, p)` qui renvoie le coefficient binomial $\binom{n}{p}$.

Par exemple, `bernoulli(10)` renvoie 0,075 757 575 757 575 76 qui est une valeur approchée de $b_{10} = \frac{5}{66}$.

V | CCINP MP 2020

Questions intercalées dans le sujet.

VI | CCINP MP 2019

Dans cet exercice « Algorithme de décomposition primaire d'un entier » (*Informatique pour tous*), on se propose d'écrire un algorithme pour décomposer un entier en produit de nombres premiers. Les algorithmes demandés doivent être écrits en langage **Python**. On sera très attentif à la rédaction et notamment à l'indentation du code.

On définit la valuation p-adique [de n] pour p nombre premier et n entier naturel non nul :

- Si p divise n, on note $v_p(n)$ le plus grand entier k tel que p^k divise n.
- Si p ne divise pas n, on pose $v_p(n) = 0$.

L'entier $v_p(n)$ s'appelle la valuation p-adique de n.

Q.1 Écrire une fonction booléenne `estPremier(n)` qui prend en argument un entier naturel non nul n et qui renvoie le booléen `True` si n est premier et le booléen `False` sinon. On pourra utiliser le critère suivant : un entier $n \geqslant 2$ qui n'est divisible par aucun entier $d \geqslant 2$ tel que $d^2 \leq n$, est premier.

Q.2 En déduire une fonction `liste_premiers(n)` qui prend en argument un entier naturel non nul n et renvoie la liste des nombres premiers inférieurs ou égaux à n.

Q.3 Pour calculer la valuation 2-adique de 40, on peut utiliser la méthode suivante :

- 40 est divisible par 2 et le quotient vaut 20.
- 20 est divisible par 2 et le quotient vaut 10.
- 10 est divisible par 2 et le quotient vaut 5.
- 5 n'est pas divisible par 2.

La valuation 2-adique de 40 vaut donc 3.

Écrire une fonction `valuation_p_adique(n,p)` **non récursive** qui implémente cet algorithme. Elle prend en arguments un entier naturel n non nul et un nombre premier p et renvoie la valuation p-adique de n. Par exemple, puisque $40 = 2^3 \times 5$, `valuation_p_adique(40,2)` renvoie 3, `valuation_p_adique(40,5)` renvoie 1 et `valuation_p_adique(40, 7)` renvoie 0.

Q.4 Écrire une deuxième fonction cette fois-ci **récursive** `val_p_adique(n,p)` qui renvoie la valuation p-adique de n.

Q.5 En déduire une fonction `decomposition_facteurs_premiers(n)` qui calcule la décomposition en facteurs premiers d'un entier $n \geqslant 2$. Cette fonction doit renvoyer la liste des couples $(p, v_p(n))$ pour tous les nombres premiers p qui divisent n.

Par exemple, `decomposition_facteurs_premiers(40)` renvoie la liste `[[2, 3], [5, 1]]`.

VII | CCINP MP 2018

Les questions sont au milieu du problème, je ne donne que les questions qui servent

On considère l'intégrale de Gauss $I = \displaystyle\int_0^1 e^{-x^2}\,dx$.

Q.1. Démontrer à l'aide d'une série entière que $I = \displaystyle\sum_{n=0}^{+\infty} \frac{(-1)^n}{(2n+1)n!}$. On pose, pour $n \in \mathbb{N}$, $s_n = \displaystyle\sum_{k=0}^{n} \frac{(-1)^k}{(2k+1)k!}$.

Q.2. Justifier que pour tout $n \in \mathbb{N}$, on

$$|I - s_n| \leq \frac{1}{(2n+3)(n+1)!}$$

Q.3. *Informatique* : écrire une fonction récursive `factorielle` qui prend en argument un entier n et renvoie l'entier $n!$.

Q.4. *Informatique* : en déduire un script qui détermine un entier N tel que $|I - s_N| \leq 10^{-6}$.

Soit $f : [a,b] \to \mathbb{R}$ une fonction continue. On se donne $n+1$ points $x_0, x_1, \ldots, x_n$ dans $[a,b]$, deux à deux distincts. On appelle polynome interpolateur de f aux points x_i, un polynôme $P \in \mathbb{R}_n[X]$ qui coïncide avec f aux points x_i, c'est-à-dire tel que pour tout $i \in [\![0;n]\!]$, $P(x_i) = f(x_i)$.

Existence du polynôme interpolateur

Pour tout entier i de $[\![0;n]\!]$, on définit le polynôme l_i de $\mathbb{R}_n[X]$ par :

$$l_i(X) = \prod_{\substack{k=0 \\ k\neq i}}^{n} \frac{X - x_k}{x_i - x_k}$$

On pose :

$$L_n(f) = \sum_{i=0}^{n} f(x_i) l_i(X)$$

Q.7. Démontrer que $L_n(f)$ est un polynôme interpolateur de f aux points x_i, puis démontrer l'unicité d'un tel polynôme.
Un tel polynôme est appelé polynôme interpolateur de Lagrange.

Calcul effectif du polynôme interpolateur de Lagrange

Q.8. *Informatique* : si $y_0, \ldots, y_n$ sont des réels, le polynôme $P = \sum_{i=0}^{n} y_i l_i(X)$ est l'unique polynôme de $\mathbb{R}_n[X]$ vérifiant $P(x_i) = y_i$ pour tout i. Écrire en langage Python une fonction `lagrange` qui prend en arguments x une liste de points d'interpolations x_i, y une liste d'ordonnées y_i de même longueur que x, a un réel, et qui renvoie la valeur de P en a.
Par exemple, si `x=[-1,0,1]` et `y=[4,0,4]`, on montre que $P = 4X^2$ et donc $P(3) = 36$. Ainsi `lagrange(x,y,3)` renverra 36.

Q.9. *Informatique* : chercher le polynôme interpolateur $P = a_0 + a_1 X + \cdots + a_n X^n$ de f aux points x_i revient aussi à résoudre le système linéaire suivant d'inconnues $a_0, \ldots, a_n$:

$$\begin{cases} P(x_0) = f(x_0) \\ \vdots \\ P(x_n) = f(x_n) \end{cases} \iff V \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ \vdots \\ f(x_)n \end{pmatrix}$$

où V est une matrice carrée de taille $n+1$.
Déterminer la matrice V et indiquer la complexité du calcul en fonction de n, lorsque l'on résout ce système linéaire par la méthode du pivot de Gauss.

VIII | CCINP MP 2017

Partie V - Informatique : calcul effectif de la probabilité invariante d'une matrice stochastique strictement positive

Si A est une matrice stochastique strictement positive, on a établi dans la partie précédente la convergence de la suite $(\mu_n)_{n\in\mathbb{N}}$ associée à la matrice A. Ceci fournit un algorithme de calcul de la probabilité invariante par A. On en propose une implémentation en langage Python. On sera très attentif à la rédaction et notamment à l'indentation du code.

Un vecteur x de $\mathbb{R}^p$ sera représenté en Python par une liste de flottants. Par exemple, le vecteur $x = (1,2,3)$ de $\mathbb{R}^3$ sera représenté par la liste `[1,2,3]`. De même, une matrice A sera représentée par une liste dont les éléments sont les lignes de la matrice. Par exemple, la matrice $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ sera représentée par la liste `[[1,2,3],[4,5,6]]`.

Q29. On exécute le script suivant `A=[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]` qui représente la matrice

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}.$$

Donner les valeurs renvoyées lorsque l'on exécute `len(A)`, `A[1]` et `A[2][1]`.

Q30. Écrire une fonction `difference` qui prend en arguments deux vecteurs x et y de même taille et renvoie le vecteur $x - y$. Par exemple si $x = (5,2)$ et $y = (3,7)$, `difference(x,y)` renverra `[2,-5]`.

Q31. Écrire une fonction `norme` qui prend en argument un vecteur $x = (x_1, \ldots, x_p)$ et renvoie sa norme infinie $\|x\|_\infty = \max\{|x_i| \mid i \in [\![1;p]\!]\}$ (on pourra utiliser librement la fonction `abs` qui renvoie la valeur absolue d'un nombre, mais on s'interdit l'utilisation de la fonction `max` déjà implémentée dans Python).

Q32. Écrire une fonction `itere` qui prend en arguments un vecteur ligne x et une matrice carrée de même taille que x et qui renvoie le vecteur xA. Par exemple si $x=(1,1)$ et $A=\begin{pmatrix}1 & 2\\ 4 & 5\end{pmatrix}$, on a $xA=(5,7)$ et donc `itere(x,A)` renverra `[5,7]`.

Q33. On a vu, dans la **Partie IV**, que si A est une matrice strictement positive, la suite de vecteurs lignes de $\mathbb{R}^p$ associée $(\mu_n)_{n\in\mathbb{N}}$ définie par la relation : $\forall n\in\mathbb{N},\ \mu_{n+1}=\mu_n A$ convergeait vers un vecteur μ_∞ indépendant du choix de μ_0 vecteur stochastique.

Écrire une fonction `probaInvariante` qui prend en arguments une matrice stochastique strictement positive A de $M_p(\mathbb{R})$ et un réel $\varepsilon>0$ et qui renvoie le premier terme μ_k de la suite $(\mu_n)_{n\in\mathbb{N}}$ avec $\mu_0=\left(\frac{1}{p},\frac{1}{p},\dots,\frac{1}{p}\right)$ tel que $\|\mu_k-\mu_{k-1}\|_\infty\leqslant\varepsilon$. On ne demandera pas à l'algorithme de vérifier que la matrice passée en argument est bien stochastique et strictement positive.

Par exemple, si $A=\begin{pmatrix}\frac{1}{2} & \frac{1}{2}\\ \frac{1}{4} & \frac{3}{4}\end{pmatrix}$ et $\varepsilon=10^{-6}$,

`probaInvariante(A,eps)` renverra `[0.33333396911621094, 0.6666660308837891]`.

IX | CCINP MP 2016

EXERCICE I : INFORMATIQUE

Les algorithmes demandés doivent être écrits en Python. On sera très attentif à la rédaction et notamment à l'indentation du code. Cet exercice étudie deux algorithmes permettant le calcul du pgcd (plus grand diviseurs communs) de deux entiers naturels.

I.1. Pour calculer le pgcd de 3705 et 513, on peut passer en revue tous les entiers $1,2,3,\cdots,512,513$ puis renvoyer parmi ces entiers le dernier qui divise à la fois 3705 et 513. Il sera alors bien le plus grand des diviseurs commun à 3705 et 513. Écrire une fonction `gcd` qui renvoie le pgcd de deux entiers naturels non nuls, selon la méthode décrite ci-dessus. On pourra éventuellement utiliser librement l'instruction `min(a,b)` qui calcule le minimum de a et b. Par exemple `gcd(3705 ,513)` renverra 57.

I.2. L'algorithme d'Euclide permet aussi de calculer le pgcd. Voici une fonction Python nommée `euclide` qui implémente l'algorithme d'Euclide.

```
def euclide(a,b):
    """Données: aet b deux entiers naturels
       Résultat: le pgcd de a et b, calculé par l'algorithme d'Euclide"""
    u=a
    v=b
    while v !=0:
        r=u%v
        u=v
        v=r
    return u
```

Écrire une fonction « récursive » `euclide_rec` qui calcule le pgcd de deux entiers naturels selon l'algorithme d'Euclide.

I.3. On note $(F_n)_{n\in\mathbb{N}}$ la suite des nombres de Fibonacci définie par :

$$F_0=0, F_1=1,\quad \forall n\in\mathbb{N},\ F_{n+2}=F_{n+1}+F_n.$$

I.3.a. Écrire les divisions euclidiennes successivement effectuées lorsque l'on calcule le pgcd de $F_6=8$ et $F_5=5$ avec la fonction `euclide`.

I.3.b. Soit $n\geqslant 2$ un entier. Quel est le reste de la division euclidienne de F_{n+2} par F_{n+1} ? On pourra utiliser librement que la suite $(F_n)_{n\in\mathbb{N}}$ est strictement croissante à partir de $n=2$. En déduire, sans démonstration, le nombre u_n de divisions euclidiennes effectuées lorsque l'on calcule le pgcd de F_{n+2} et F_{n+1} avec la fonction `euclide`.

I.3.c. Comparer pour n au voisinage de $+\infty$, ce nombre u_n, avec le nombre v_n de divisions euclidiennes effectuées pour le calcul du pgcd de F_{n+2} et F_{n+1} par la fonction `gcd`. On pourra utiliser librement que F_n est équivalent au voisinage de $+\infty$, à $\phi^n/\sqrt{5}$ où $\phi=(1+\sqrt{5})/2$ est le nombre d'or.

I.4. Écrire une fonction `fibo` qui prend en argument un entier naturel n et renvoie le nombre de Fibonacci F_n. Par exemple, `fibo(6)` renverra 8.

I.5. En utilisant la fonction `euclide`, écrire une fonction `gcd_trois` qui renvoie le pgcd de trois entiers naturels. Par exemple, `gcd_trois(18, 30, 12)` renverra 6.

X | CCINP MP 2015

EXERCICE I : INFORMATIQUE

Les algorithmes demandés doivent être écrits en Python. On sera très attentif à la rédaction et notamment à l'indentation du code.

Voici, par exemple, un code Python attendu si l'on demande d'écrire une fonction nommée `maxi` qui calcule le plus grand élément d'un tableau d'entiers :

```
def maxi(t):
    """Données: t un tableau d'entiers non vide
       Résultat: le maximum des éléments de t"""
    n =len(t) # la longueur du tableau t
    maximum = t[0]
    for k in range(1,n):
        if t[k] > maximum:
            maximum = t[k]
    return maximum
```

L'instruction `maxi([4,5,6,2])` renverra alors 6.

I.1. Donner la décomposition binaire (en base 2) de l'entier 21.

On considère la fonction mystere suivante :

```
def mystere(n, b):
    """Données: n > 0 un entier et b > 0 un entier
       Résultat: ......."""
    t = [] # tableau vide
    while n > 0:
        c = n % b
        t.append(c)
        n = n // b
    return t
```

On rappelle que la méthode `append` rajoute un élément en fin de liste. Si l'on choisit par exemple `t = [4,5,6]`, alors, après avoir exécuté `t.append(12)`, la liste `t` a pour valeur `[4,5,6,12]`.

Pour $k \in \mathbb{N}*$, , on note c_k , t_k et n_k les valeurs prises par les variables `c`, `t` et `n` à la sortie de la k-ème itération de la boucle `while`.

I.2. Quelle valeur est renvoyée lorsque l'on exécute `mystere(256,10)` ?

On recopiera et complétera le tableau suivant, en ajoutant les éventuelles colonnes nécessaires pourtracer entièrement l'exécution.

k	1	2	$\cdots$
c_k			$\cdots$
t_k			$\cdots$
n_k			$\cdots$

I.3. Soit $n > 0$ un entier. On exécute `mystere(n,10)`. On pose $n_0 = n$.

I.3.a Justifier la terminaison de la boucle `while`.

I.3.b On note p le nombre d'itérations lors de l'exécution de `mystere(n,10)`. Justifier que pour tout $k \in [\![0;p]\!]$, on a $n_k \leqslant \frac{n}{10^k}$. En déduire, une majoration de p en fonction de n.

I.4. En s'aidant du script de la fonction `mystere`, écrire une fonction `somme_chiffres` qui prend en argument un entier naturel et renvoie la somme de ses chiffres. Par exemple, `somme_chiffres(256)` devra renvoyer 13.

I.5. Écrire une version récursive de la fonction `somme_chiffres`, on la nommera `somme_rec`.