Introduction à l’algorithmique

Master 1 Bio-Informatique, Université de Bordeaux

2020-2022 - Version 0.41 - En cours d’écriture

Sous licence libre Creative Commons BY-NC-SA 4.0

Table des matières

1 Licence **4**

2 Fonctionnement d’un ordinateur **4**
2.1 Architecture d’un ordinateur . . . 4
2.2 La mémoire . . . 6
2.3 Programme . . . 7
2.4 L’algorithmique . . . 11

3 Le langage python **12**
3.1 Utilisation de Pythontutor pour maîtriser l’utilisation de la mémoire . . . 12
3.2 Variable et références . . . 12
3.3 Passage par références . . . 13
3.4 Exercice . . . 13

4 Interface et bibliothèque **15**
4.1 Bibliothèque . . . 15
4.2 Prototypes et implémentions multiples . . . 16

5 Pile et File **19**
5.1 Les piles (LIFO) . . . 19
5.2 Les files (FIFO) . . . 21

6 La récursivité **24**
6.1 La pile d’exécution . . . 24
6.2 La récursivité . . . 28

7 Recherche d’un élément dans une tableau **30**
7.1 Recherche linéaire . . . 30
7.2 Recherche dichotomique . . . 30
7.3 Estimer le nombre d’instructions de l’algorithme de recherche dichotomique (optionnel-à regarder à la fin de la séance) . . . 31

8 Listes **31**
8.1 Listes chaînées ou listes simplement chaînées . . . 32
8.2 Listes doublement chaînées . . . 34

9 La Complexité **37**
9.1 Présentation du problème 37
9.2 La complexité en pire cas et en meilleur cas 38
9.3 Comparer les algorithmes en utilisant des comparaisons asymptotiques 39
9.4 Règles de calculs pour $\mathcal{O}$ 39
9.5 Quelques exemples 40
9.6 Complexité en moyenne et complexité amortie 40

10 Algorithmes de Tri **40**
10.1 Tri à bulles 40
10.2 Tri par insertion (tri du joueur de carte) 41
10.3 Tri par sélection 41
10.4 Tri rapide 42
10.5 Tri fusion 43

11 Arbres **43**
11.1 Arbres 43
11.2 Exemples d'arbres en informatique 47
11.3 Une API pour les arbres 49
11.4 Arbres binaires 51
11.5 Une API pour les arbres binaires 55
11.6 Quelques définitions pour les arbres et les arbres binaires 57
11.7 Parcours d'un arbre 58
11.8 Arbres binaires de recherche 62

12 Les graphes **68**
12.1 Ensembles, multiensembles, listes et mots 68
12.2 Définition des graphes 70
12.2.1 Les graphes non orientés 70
12.2.2 Les graphes orientés 71
12.2.3 Les graphes étiquetés (orientés et non orienté) 73
12.2.4 Arêtes et arcs multiples, cas particuliers et notations abusives 73
12.2.5 Chemins et chaînes 74
12.2.6 Accessibilité, connexité, forte conexité, composantes connexes et composantes fortement connexes 76
12.2.7 Cycles, circuits et arbres 78
12.2.8 Degré d'un sommet et degré total d'un graphe 80
12.3 Implémentation des graphes 81
12.3.1 Matrice d'adjacence 81
12.3.2 Liste d'arcs adjacents et liste d'arêtes adjacentes 85
12.3.3 Liste d'adjacence (liste de sommets adjacents) 86
12.3.4 Matrice d'incidence 87
12.3.5 Avantages et inconvénients des différentes structures de données de graphe 88
12.4 Parcours en Largeur et parcours en profondeur 88
12.4.1 Parcours en Largeur 88
12.4.2 Parcours en profondeur 88
12.5 Algorithmes de plus court chemin 89
12.6 Dijkstra 89
12.6.1 Bellman 89
12.7 Tri topologique et calcul de composantes fortement connexes 89
12.7.1 Tri topologique 89
12.7.2 Composantes fortement connexes 90

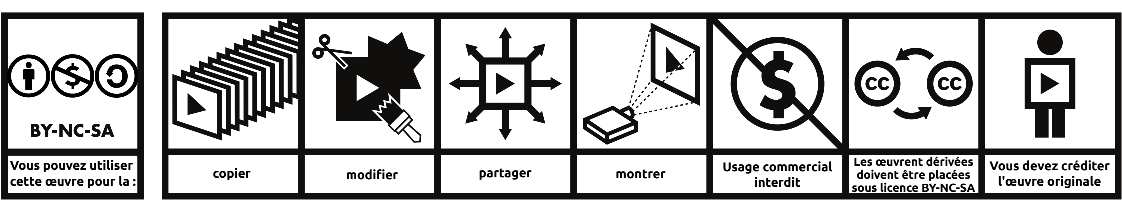
13 Références **91**

A Le langage de programmation Python **92**
A.1 Écrire et exécuter un programme écrit en Python 92
A.2 Les variables et les affectations 92
A.2.1 Les variables 92
A.2.2 Afficher le contenu d'une variable 93
A.2.3 Copier le contenu de la mémoire 93
A.3 Les commentaires 94
A.4 Les entiers, les réels, les booléens et les opérations arithmétiques 94
A.5 Les fonctions 95
A.6 Les saut conditionnels 97
A.7 Les boucles 99
A.8 Les listes et les tableaux 99
A.9 Les tuples 101
A.10 Le texte et les chaînes de caractères 101

B Quelques calculs pour la complexité **102**
B.1 Quelques symboles classiques 102
B.2 Les sommes et les produits 102
B.3 Les binomiaux 103
B.4 Calcul de sommes classiques 104

1 Licence

Ce cours est sous licence libre Creative Commons BY-NC-SA dans sa version 4.0. Les droits et devoirs associés à cette licence sont résumés dans l'image suivante qui a été conçue originallement [1] par Piotrek Chuchla.



Attribution - Pas d'utilisation commerciale - Partage dans les mêmes conditions 4.0 – Cette licence autorise autrui à copier, modifier, partager votre œuvre, sauf pour des utilisations commerciales. Les personnes utilisant votre œuvre s'engagent à la créditer, à intégrer un lien vers la licence et à indiquer si des modifications ont été effectuées à l'œuvre. Si elles réalisent des modifications, l'œuvre dérivée devra être diffusée sous licence CC-BY-NC-SA également. **https://creativecommons.org/licenses/by-nc-sa/4.0/deed.fr**

2 Fonctionnement d'un ordinateur

2.1 Architecture d'un ordinateur

Un *ordinateur* est un système constitué d'une unité de calcul et d'une mémoire. La *mémoire* contient un ensemble de données codées par une suite de 1 et de 0. L'*unité de calcul* est la partie de l'ordinateur qui permet de lire, écrire et modifier la mémoire. La figure 1 montre cette architecture de l'ordinateur.

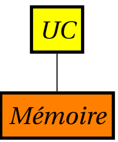


FIGURE 1 – Architecture d'un ordinateur

Il existe différentes façons de réaliser un ordinateur. Par exemple, la machine de Turing (que nous ne présenterons pas dans ce cours), représentée à la figure 2, est un modèle purement théorique d'un ordinateur.

1. L'œuvre originale, sous licence CC-BY 3.0, est de Piotrek Chuchla et est accessible à l'adresse `https://www.behance.net/gallery/5783221/Creative-Commons-poster` (accédé le 24 février 2020). Cette œuvre a elle même été traduite et adaptée par Pascal Combemorel, sous licence CC-BY 4.0, et est accessible à l'adresse `https://upload.wikimedia.org/wikipedia/commons/4/49/Poster_Creative_Commons.svg` (accédé le 24 février 2020)

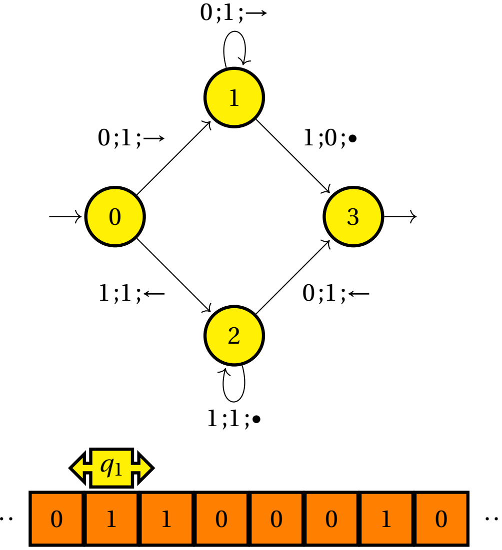


FIGURE 2 – Exemple d'une machine de Turing

Un modèle théorique d'un ordinateur est une représentation abstraite d'un ordinateur d'écrite à l'aide du langage mathématique. Dans la pratique, il n'est pas utilisé pour réaliser des programmes. Il est utilisé pour démontrer qu'un problème peut être résolu par un ordinateur.

Les téléphones portables, les ordinateurs de bureaux, les ordinateurs portables, les consoles de jeux sont des exemples d'ordinateurs.

Dans les ordinateurs modernes, l'unité centrale et la mémoire sont situées dans différents endroits. L'ordinateur est aussi accompagné de nombreux *périphériques* : claviers, écrans, enceintes, afin de permettre aux utilisateurs d'interagir avec l'ordinateur. La figure 3 montre l'architecture d'un ordinateur actuel :

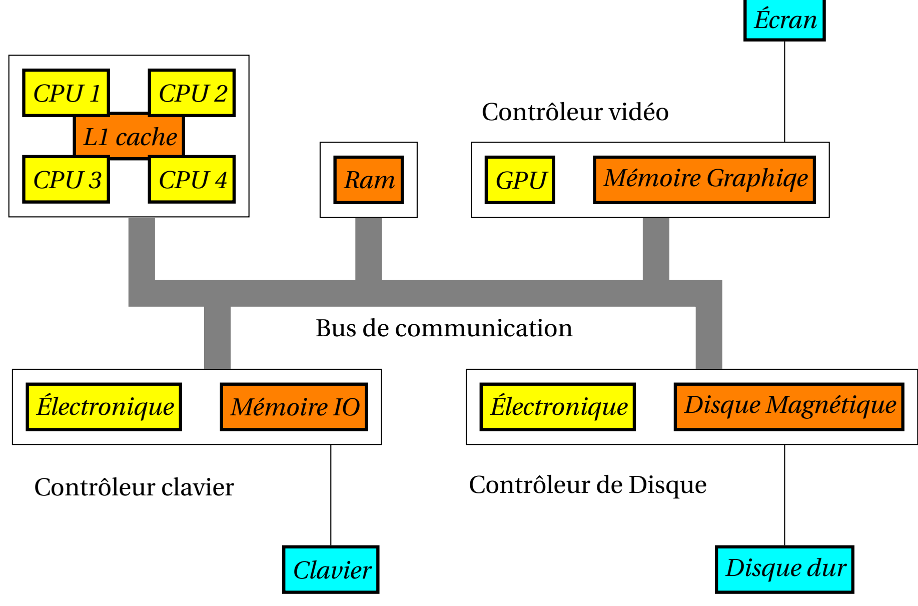


FIGURE 3 – Architecture simplifiée d'un ordinateur

Dans la figure 3, les boîtes oranges représentent la mémoire, les boîtes jaunes les unités de calculs et les boîtes bleues les périphériques. Dans un ordinateur, le système d'exploitation gère les unité de calcul et les mémoires de sorte que, pour le programmeur, tout se passe comme s'il y avait une unique mémoire et une unique unité de calcul, comme cela est présentée dans la figure 1.

Des ordinateurs, il y a en plein partout, en électronique on parle de micro-controlleur. Un exemple de micro-controlleur est le stm32F4 dont vous pouvez, juste par curiosité, trouver son architecture en tapant, dans un moteur de recherche les mots clefs suivant : stm32f4, datasheet et pdf. Vous pouvez trouvez son architecture à la figure "block diagram". Sachez que, malgré l'extrême complexité de ces documentations techniques, ces processeurs se programment en *C* mais aussi depuis 2015 en python : `http://micropython.org/`. Sur ce site, vous y trouverez un exemple de micro-controlleur habillé dans le plus simple appareil. ;)

2.2 La mémoire

La *mémoire* d'un ordinateur est une suite de 0 et de 1.

Voici à quoi ressemble un fragment de mémoire :

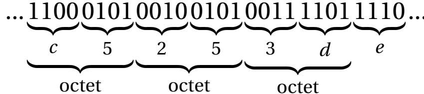
...110001010010010100111101110...

Chaque entier de cette suite est appelé un bit (en anglais : bit). Il désigne la quantité élémentaire d'information en informatique.

Généralement, en informatique, on regroupe les bits par groupe de 8 que l'on appelle *octet* (en anglais : Byte (avec un grand B). Chaque octet est décomposé en 2 groupes de 4 bits, (0000, 0001, 0010, $\ldots$, 1111) qui est représenté par une lettre parmi $0, 1, 2, \ldots, 9, a, b, c, d, e, f$, où la correspondance, appelée code hexadécimal, est donnée par :

$$\begin{array}{ll} 0000 \Longrightarrow 0 & 1000 \Longrightarrow 8 \\ 0001 \Longrightarrow 1 & 1001 \Longrightarrow 9 \\ 0010 \Longrightarrow 2 & 1010 \Longrightarrow a \\ 0011 \Longrightarrow 3 & 1011 \Longrightarrow b \\ 0100 \Longrightarrow 4 & 1100 \Longrightarrow c \\ 0101 \Longrightarrow 5 & 1101 \Longrightarrow d \\ 0110 \Longrightarrow 6 & 1110 \Longrightarrow e \\ 0111 \Longrightarrow 7 & 1111 \Longrightarrow f \end{array}.$$

Le fragment de mémoire donné précédemment ce code ainsi, en hexadécimal :



ce qui donne :

$$\ldots\ c5\ 25\ 3d\ e\ldots$$

On décrit les tailles des différentes mémoires à l'aide de cette unité (l'octet).

> **Exercice 1**
> Convertissez en hexadécimal le code suivant :
>
> 00101100101000101011111100001001.

Ainsi,

— un disque dur contient généralement entre 100 Go et 400 Go ;

— une clé usb contient entre 1 et 32 Go ;

— une mémoire vive (RAM : Random Access Memory) contient 1 à 4 Go;
— la mémoire cache du processeur contient (1 à 4 Mo);
— la taille d'un film en définition normale est d'environ 700 Mo;
— la taille d'une musique est d'environ 3 Mo;
— les registres mémoires d'un processeur contiennent 4 ou 8 octets (32 bits ou 64 bits) selon l'architecture du processeur.

Un *registre de processeur* est un emplacement mémoire interne au processeur. Il sert à réaliser des calculs et des transferts de données. Pour donner une image (grossière), son équivalent en cuisine est le "verre de cuisine" : c'est le plus petit "récipient" disponible pour transférer et mesurer les quantités d'ingrédient à utiliser pour faire une recette.

Dans ce cours, on appellera *petit espace mémoire* un espace mémoire de la taille d'un registre de processeur. Sa taille est de 32 ou 64 bits (4 ou 8 octets) selon l'architecture.

2.3 Programme

On appelle programme l'ensemble des instructions qui doivent être exécutées par l'unité de calcul (processeur) de l'ordinateur.

Voici un exemple de programme écrit en Python :

```
print( "Bonjour" )
print( "L'ordinateur va compter jusqu'à 4 :" )
for i in range(4):
    print( "numero : " + str(i+1) )
```

On appelle *langage* les règles d'écritures de ces inscriptions.

Il existe beaucoup de langages :

— Le langage machine
— L'assembleur
— Le C
— Python
— Java
— etc ...

Le langage machine est un langage lisible directement par le processeur. Il est très peu compréhensible pour un humain normal.

Par exemple, voici un exemple de langage machine :

```
457f 464c 0102 0001 0000 0000 0000 0000
0002 003e 0001 0000 04c0 0040 0000 0000
0040 0000 0000 0000 1168 0000 0000 0000
0000 0000 0040 0038 0009 0040 001f 001c
0006 0000 0005 0000 0040 0000 0000 0000
0040 0040 0000 0000 0040 0040 0000 0000
01f8 0000 0000 0000 01f8 0000 0000 0000
0008 0000 0000 0000 0003 0000 0004 0000
0238 0000 0000 0000 0238 0040 0000 0000
0238 0040 0000 0000 001c 0000 0000 0000
...
```

L'assembleur et le C sont deux langages, compréhensible par l'utilisateur, mais pas par la machine.

Voici, par exemple, le programme précédent (écrit en langage machine), mais écrit maintenant en assembleur :

```
    .file "essai.c"
    .section .rodata
.LC0:
    .string "Bonjour"
    .text
    .globl main
    .type main, @function
main:
.LFB0:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq %rsp, %rbp
    .cfi_def_cfa_register 6
    movl $.LC0, %edi
    call puts
    movl $0, %eax
    popq %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size main, .-main
    .ident "GCC: (GNU) 4.6.0"
    .section .note.GNU-stack,"",@progbits
```

Voici, maintenant, toujours le même programme, mais écrit en C :

```
#include <stdio.h>
int main(){
    printf("Bonjour\n");
    return 0;
}
```

L'ordinateur exécuté sans problèmes les instructions du tout premier programme, puisqu'il est écrit en langage machine.

Par contre, ce n'est pas le cas des deux derniers.

Pour que l'ordinateur puisse réaliser les instructions des deux derniers programmes, il faut convertir ces programmes en langage machine, avant de les exécuter. C'est à dire transformer les deux derniers pour obtenir le tout premier programme.

Pour ce faire, on utilise un compilateur. Un compilateur est un programme qui prends le code source écrit dans un langage compréhensible par un humain et le convertit intégralement en un programme écrit en langage machine, que l'on appelle généralement exécutable.

Par exemple, les développeurs d'un jeux vidéos, écrivent le jeu dans un langage donné (le C/C++ par exemple), le compile ensuite et ils diffusent aux joueurs l'exécutable (en langage machine) pour qu'il puisse démarrer le jeu et jouer.

Dans ce cours, nous allons utiliser le langage de programmation *python*.

Voici la traduction en python des programmes précédent :

```
print("Bonjour")
```

Le code python ne se compile pas, pour exécuter les instructions d'un langage écrit en python, il faut utiliser un interpréteur. Un interpréteur est un programme qui lit les instructions écrits en python,

les convertit en instructions processeurs (langage machine) et fait exécuter ces instructions, à la volée au fur et à mesure de la lecture du code source.

La figure 4 résume le processus qui permet d'exécuter les instructions d'un programme par un ordinateur.

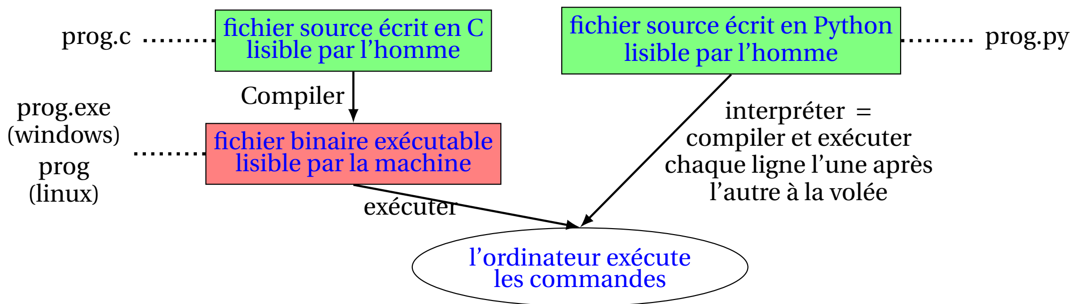


FIGURE 4 – Compilation/exécution vs interprétation d'un code source

Pour bien comprendre le processus de la figure 4, effectuez les exercices suivants :

Exercice 2: Compiler et exécuter un programme en C

Créer un fichier fichier `prog.c` à l'aide de gedit (en texte brut - codage ASCII - sans utilise librof-fice) et contenant

```
#include <stdio.h>
int main(){
    printf("Bonjour\n");
    printf("Ceci est mon premier programme\n");
    printf("2 + 3 = %d \n", 3+2 );
    return 0;
}
```

Ouvrez un terminal et compiler le programme à l'aide de la ligne de commande suivante :

```
gcc prog.c –o prog
```

Ouvrez à l'aide de gedit le programme `prog` obtenu. Regardez ensuite directement son code hexadécimal en tapant :

```
hexedit prog
```

Vérifiez que le code est illisible car c'est du code machine.
Exécutez le programme en tapant en ligne de commande :

```
./prog
```

Vous venez de créer, de manière simplifié un programme. Cette façon de procéder (en plus compliqué) est très utilisé dans la communauté des développeurs d'outils Gnu/Linux, ou dans la communauté des développeurs de jeux vidéos.

Exercice 3: Interpréter un programme python

Créer un fichier fichier `prog.py` à l'aide de gedit (en texte brut - codage ASCII - sans utiliser librofiice) et contenant

```
print( "Bonjour" )
print( "Ceci est mon premier programme" )
print( "2 + 3 = " + str(3+2) )
```

Ouvrez un terminal et exécutez directement le script :

```
python3 prog.py
```

Vous venez de créer, de manière simplifié un programme. Cette façon de procéder (en plus compliqué) est très utilisé dans la communauté scientifique, mais aussi en ingénierie quand il faut développer rapidement des fonctionnalités, il est aussi largement utilisé pour les administrateurs réseaux et système.

Exercice 4: Interraction manuelle avec ipython3

Démarrez l'interpréteur python `ipython3` en tapant la ligne de commande :

```
ipython3
```

Vous pouvez maintenant écrire et exécutez chaque ligne du programme de l'exercice **??** en les tapant manuellement . Vous obtiendrez ceci :

```
In [1]: print( "Bonjour" )
Bonjour

In [2]: print( "Ceci est mon premier programme" )
Ceci est mon premier programme

In [3]: print( "2 + 3 = " + str(3+2) )
2 + 3 = 5
```

Essayez aussi les commandes suivantes qui sont propres à l'interpréteur ipython3 :

```
l = [3,4,5]
help(l)
l?
help( l.append )

pwd
path = _
print(path)
cd /tmp
ls
cd $path # Attention, ceci n'est pas du python, c'est spécifique àl'interpréteur ipython3
ls
```

Vous pouvez exécutez tous le programme d'un coup à l'aide de la commande :

```
%load prog.py
```

Vous pouvez modifier le programme en tapant (cela provoquera automatiquement son exécution quand vous quitterez l'éditeur)

```
%edit prog.py
```

ajouter à la fin le code :

```
var = [42, 3, 2]
```

Tapez maintenant :

```
var
```

La variable est accessible et vous pouvez maintenant la manipuler à votre guise.
Ainsi, vous voyez qu'il n'est pas nécessaire de faire des menus, et des interfaces graphiques pour pouvoir interagir avec les calculs réalisés par un ordinateur. C'est pour cela que les langages interprétés sont très utilisés par les administrateurs systèmes ou par les scientifiques qui cherchent à faire des expérimentations calculatoires sans avoir à s'occuper de gérer une interface graphique ou une interface utilisateur.
Par exemple, utilisez ipython3 comme une super calculatrice en exécutant ipython3, puis en tapant le code suivant :

```
from math import *
sqrt(5)
resultat = pi + sqrt(3) - 2**3
resultat
```

Ambiancez-vous et essayez le code suivant, pour réaliser à quel point il est pratique d'utiliser des langages interprétés. Nous n'expliquerons pas le code réalisé, qui utilise une bibliothèque de calcul scientifique.

```
import numpy # Bibliothèque de calcul scientifique
from matplotlib import pyplot

X = numpy.arange(0,2, 0.01) # Tableaux de réels de 0 à2
                            # par pas de 0.01
Y1 = numpy.sqrt(x) # Attention ! X n'est pas une
                   # liste python classique !
                   # Mais un tableau numpy.
Y2 = X*X + X - 1 # Idem, X n'est pas un liste python
                   # classique ou cette opération serait
                   # illégale.
pyplot.plot(X, Y1)
pyplot.plot(X, Y2)
pyplot.show()
```

Pour finir, sachez qu'utiliser un langage interprété à un coût : les temps d'exécutions des langages interprétés sont plus longs. Ainsi un code C est généralement 1000 fois plus rapide que le même code écrit en python. Cela explique pourquoi les développeurs de gros jeux vidéos continuent à utiliser le C/C++ pour écrire leurs logiciels. En fait, la réalité est légèrement plus complexe. Les jeux sont des applications hybrides ou le coeur est programmée en C/C++ pour la vitesse d'exécution et le scénario et les règles du jeux, qui ne nécessitent pas une vitesse d'exécution rapide, sont eux scriptés et écrits dans un autre langage interprété comme “Lua” ou “Python” par exemple.

2.4 L'algorithmique

Un *algorithme* est une suite finie et non-ambiguë d'opérations ou d'instructions permettant de résoudre un problème.

Le mot algorithme vient du nom latinisé du mathématicien arabo-musulman Al-Khawarizmi, surnommé "le père de l'algèbre". Le domaine qui étudie les algorithmes est appelé l'algorithmique.

L'*algorithmique* est l'ensemble des règles et des techniques qui sont impliquées dans la définition et la conception d'algorithmes, c'est-à-dire de processus systématiques de résolution d'un problème permettant de décrire les étapes vers le résultat.

3 Le langage python

Dans ce cours, nous allons utiliser le langage de programmation Python pour étudier différentes structures algorithmiques et différents algorithmes qui sont aux fondements de la discipline.

Nous supposons que le lecteurs a déjà suivi un cours de programmation de python. Si ce n'est pas le cas, le lecteur trouvera à l'annexe A une introduction en la matière.

Dans cette section, nous allons mettre en valeur quelques conseils et détails techniques qu'il faut impérativement maîtriser pour pouvoir faire les exercices et comprendre les exemples de ce cours.

3.1 Utilisation de Pythontutor pour maîtriser l'utilisation de la mémoire

Il est impératif de maîtriser le fonctionnement de la mémoire et la gestion par python de celle-ci.

Pour vous aider à parfaire vos connaissance ou à les vérifier, nous vous conseillons d'utiliser le site *pythontutor* accessible à l'adresse : `http://pythontutor.com/visualize.html` .

Ce site permet d'exécuter et visualiser l'évolution de la mémoire d'un programme python simple.

3.2 Variable et références

En python, une variable est le nom donné à un petit espace mémoire. Par exemple, si vous écrivez :

```
valeur = 3
```

alors, en mémoire, il sera alloué un petit espace mémoire qui aura pour nom "valeur" et dont son contenu sera le codage en binaire de 3. Utilisez PythonTutor pour vérifier cela.

La variable "valeur" est le nom d'un petit espace mémoire(de la taille d'un registre processeur) ! il n'est donc pas possible de mettre deux ou trois réels dans son espace mémoire. Ainsi, si vous écrivez maintenant :

```
valeur = [3, 4, 5]
```

alors, "valeur" reste toujours le nom d'un petit espace mémoire. Mais cette fois-ci, l'espace mémoire associé à "valeur" contient la référence vers un autre objet en mémoire (qui n'est pas dans l'environnement du programme principal) beaucoup plus grand où se trouve 3 éléments consécutifs : $[3,4,5]$. Une référence contient la position dans la mémoire des données stockées. On appelle aussi cette position l'adresse en mémoire.

Utilisez PythonTutor pour vérifier cela, comment PythonTutor symbolise-t-il une adresse ? La réponse est donc cette note de bas de page :[2].

Ainsi, si vous écrivez :

```
valeur = [3, 4, 5]
p = valeur
p[1] = 42
print(valeur)
```

alors "p" est le nom d'un petit espace mémoire et son contenu est la copie bit à bit de l'espace ayant pour nom "valeur" à savoir une adresse : la position où se trouve la liste en mémoire.

Il est important de comprendre que vous n'avez pas copié la liste ! On dit que "p" et "valeur" pointe vers la même liste. Ainsi, l'affichage obtenu sur le terminal de programme précédent donne $[3,42,5]$.

Utilisez PythonTutor pour vérifier cela.

2. Il le symbolise par une flèche

3.3 Passage par références

Voici un exemple d'utilisation de deux fonctions en python

```
def f(x):
  tmp = x
  tmp = 5
  print(tmp)
def g( l ):
  tmp = l
  tmp[0] = 4
  print(tmp)
v = [1,2,3]
n = 4
f(n)
f(v)
g(v)
print( n )
print( v )
```

Utilisez PythonTutor pour voir comment python exécute ce code.

Lorsque l'on exécute une fonction en python, un espace mémoire spécifique est crée pour la fonction. Cet espace mémoire contient les petits espaces mémoires associées aux paramètres, et tous les petits espaces mémoires associés aux variables que vous allez créer dans le corps de la fonction.

L'initialisation des paramètres se fait par copie bit à bit du contenu du petit espace mémoire des variables qui ont été passées en paramètre à la fonction. Ainsi, quand la variable passé en paramètre contient une référence vers un objet, comme une liste par exemple, l'objet n'est pas copié, c'est l'adresse qui est copié! C'est ce qui se passe dans l'exemple précédent où la fonction g finit par modifier le contenu de la liste v en utilisant la référence passée en paramètre.

Si vous voulez faire une copie, dite copie profonde, des paramètres, vous devez utilisez la bibliothèque `copy` de la façon suivante :

```
from copy import deepcopy

def g( l ):
  tmp = deepcopy(l)
  tmp[0] = 4
  print(tmp)
v = [1,2,3]
g(v)
print( v )
```

Testez le programme précédent avec PythonTutor .

3.4 Exercice

Exercice 5
Pour chacun des programmes suivant, vous essayerez de prévoir l'exécution du programme et vous essayerez de dessiner la mémoire de la même manière que PythonTutor la dessine. Vous ferez cela SANS utiliser PythonTutor .
Une fois fait, vous vérifierez le résultat obtenu en utilisant PythonTutor .

Methodologie : Il est TRÈS important de n'exécuter PythonTutor , qu'une fois les dessins de la mémoire et la prévision d'exécution faite. Prenez le temps de donner une réponse qui soit complètement logique et TOUJOURS reproductible. Cette étape est importante, car, en cas d'échec, vous pourrez faire une introspection sur votre raisonnement et comprendre pourquoi vous vous êtes trompé. Si vous ne faite pas cela, alors, l'introspection ne sera pas possible et vous n'aurez aucune possibilité pour corriger vos erreurs de raisonnement que vous conserverez.

Voici les programmes à analyser.

Programme 1 :

```
r = [1,2,3]
p = r
p[0] = 30
print( r )
print( p )
```

Programme 2 :

```
r = 1
p = r
p = 30
print( r )
print( p )
```

Programme 3 :

```
r = [[1,2,3], [4,5,6], [7,8,9]]
p = r[0]
print( r )
print( p )
r[0][2] = 42
p[0] = 22
print( r )
print( p )
```

Programme 4 :

```
r = [1, 2, 3]
p = r[0]
print( r )
print( p )
r[0] = 42
p = 22
print( r )
print( p )
```

Programme 5 :

```
r = [1, 2, 3]
def f(x):
  x = 42
  print(x)
f(r)
print(r)
```

Programme 6 :

```
r = [1, 2, 3]
def f(r):
  r = 42
  print(r)
f(r)
print(r)
```

Programme 7 :

```
r = [1, 2, 3]
def f(x):
  x[0] = 42
  print(x)
f(r)
print(r)
```

Programme 8 :

```
r = [1, 2, 3]
def f(r):
  r[0] = 42
  print(r)
f(r)
print(r)
```

Programme 9 :

```
r = [1, 2, 3]
def f(x):
    x[0] = 42
    print(x)
f(r)
print(r)
```

Programme 10 :

```
r = [1, 2, 3]
def f(r):
    r[0] = 42
    print(r)
f(r)
print(r)
```

Programme 11 :

```
r = 1
def f(x):
    x = 42
    print(x)
f(r)
print(r)
```

Programme 12 :

```
r = 1
def f(r):
    r = 42
    print(r)
f(r)
print(r)
```

Exercice 6: DIFFICILE - A faire en fin de séance

Construire, en python, une structure de données l, utilisant les listes telle que l'on puisse écrire :

$$l\underbrace{[0][0][0][0][0]\dots[0]}_{n \text{ fois}}[1]$$

où n est arbitraire, sans produire d'erreur à l'exécution !

4 Interface et bibliothèque

4.1 Bibliothèque

En informatique, il existe de nombreuses bibliothèques. Il s'agit d'un ensemble de fichiers contenant le code de nombreuses fonctions.

Les bibliothèques sont utiles, car elle permettent à leurs utilisateurs de ne pas réimplémenter des outils déjà existant. On a déjà vu précédemment, les bibliothèques : copy, math, numpy et matplolib.

En python, tout fichier est une bibliothèque qu'il est possible d'importer.

Par exemple, si vous créez le fichier `mathemathiques.py`, contenant :

```
def somme(x,y):
    """
    Retourne la somme de deux entiers

    x : float
        Le premier terme de la somme
    y : float
        Le deuxième terme de la somme
    return : float
    """
    return float(x+y)

def produit(x,y):
```

```
    """
    Retourne la produit de deux réels

    x : float
        Le premier réel de la somme
    y : float
        Le deuxième réel de la somme
    return : float
    """
    return float(x*y)
```

Alors, dans une autre programme, que vous nommerez `test.py`, vous pouvez importer les fonctions de cette bibliothèque en tapant :

```
import mathemathiques as tools

a = tools.somme(3,2)
print(a)
b = tools.produit(3,2)
print(b)
```

4.2 Prototypes et implémentions multiples

On appelle API (Application Programming Interface) d'une bibliothèque, la définition des différentes fonctions qui la compose ainsi que la description de leurs fonctionnalités.

Par exemple, la bibliothèque précédente a pour API :

```
def somme(x,y):
    """
    Retourne la somme de deux entiers

    x : float
        Le premier terme de la somme
    y : float
        Le deuxième terme de la somme
    return : float
    """
    return NotImplemented

def produit(x,y):
    """
    Retourne la produit de deux réels

    x : float
        Le premier terme de la somme
    y : float
        Le deuxième terme de la somme
    return : float
    """
    return NotImplemented
```

A partir d'une API, on peut proposer plusieurs implémentation d'une bibliothèque.

Par exemple, la bibliothèque `gogogadgeto.py` suivante, implémente le même prototype que `mathemathique.py`

```
def somme(x,y):
    """
    Retourne la somme de deux réels

    x : float
        Le premier terme de la somme
    y : float
        Le deuxième terme de la somme
    return : float
    """
    return float(x+1+y-1)

def produit(x,y):
    """
    Retourne la produit de deux réels

    x : float
        Le premier réel de la somme
    y : float
        Le deuxième réel de la somme
    return : float
    """
    return float(x*2*y/2)
```

L'intérêt d'avoir recours à plusieurs implémentation de bibliothèque, c'est que l'on peut, dans un programme, passer d'une implémentation à une autre en changeant une seule ligne de code. Par exemple, le programme suivant :

```
import gogogadgeto as tools

a = tools.somme(3,2)
print(a)
b = tools.produit(3,2)
print(b)
```

donne le même résultat que celui utilisant la bibliothèque mathématique.

En informatique nous utilisons les bibliothèques car cela permet de factoriser le code, de tester facilement l'efficacité de différentes implémentions, d'utiliser différentes implémentation selon différents contextes.

Une bibliothèque est correctement utilisée et correctement implémentée, si l'on peut la remplacer par une autre qui implémenter le même prototype sans changer le fonctionnement du programme.

Par exemple, voici une bibliothèque `titine.py` :

```
def creer_vehicule(nb_roues):
    """
    retourne un vehicule avec un nombre de roues donné

    nb_roues : int
        le nombre de roues du véhicule
    """
    return {"nb_roues": nb_roues}
def nb_roues(vehicule):
    """
    renvoie le nombre de roues d'un véhicule donné
```

```
    return : int
    """
    return voiture["nb_roues"]
```

et voici une mauvaise utilisation de la bibliothèque `titine.py` :

```
import titine as vehicule

voiture = vehicule.creer_vehicule(4)
print( "Le véhicule a " + str(voiture["nb_roues"]) + " roues." )
```

En effet, si, à la place de `titine.py`, on utilise l'implémentation `auto.py` suivante :

```
def creer_vehicule(nb_roues):
    """
    retourne un vehicule avec un nombre de roues données

    nb_roues : int
      le nombre de roues du véhicule
    """
    return [nb_roues]
def nb_roues(vehicule):
    """
    renvoie le nombre de roues d'un véhicule donné

    return : int
    """
    return voiture[0]
```

Alors le programme que l'on a écrit devient :

```
import auto as vehicule

voiture = vehicule.creer_vehicule(4)
print( voiture["nb_roues"] )
```

et il ne fonctionne plus ! Une bonne utilisation de ces deux bibliothèques consiste à utiliser uniquement les fonctions proposées par leur API. On obtient alors le programme suivant :

```
import auto as vehicule

voiture = vehicule.creer_vehicule(4)
print( vehicule.nb_roues(voiture) )
```

Dans ce cas, que vous utilisez `titine.py` ou `auto.py`, le programme continue de fonctionner, alors que vous n'avez changé qu'une seule ligne de code.

L'algorithmique, c'est l'étude des structures de donnée et de leurs algorithmes associés. Nous allons tout au long de ce cours, définir de nouvelles structure de données et de nouvelles façon de les utiliser en proposant à chaque fois des prototypes de fonction (dans des API) qui vont décrire des fonctionnalités et des concepts informatiques pour résoudre des problèmes génériques.

Pour cela, on implémentera tous ces algorithmes et structures dans des bibliothèques où les fonctions pourront être étudiés et comparés ensemble, tout comme l'on a comparé `titine.py` et `auto.py` à l'aide du programme `test.py`.

5 Pile et File

5.1 Les piles (LIFO)

Imaginons la manipulation d'un tube de comprimés. Le tube est opaque et il n'est pas possible de voir son contenu. Le tube contient plusieurs comprimés numérotés portant éventuellement le même numéro. Lorsque l'on manipule le tube et ses comprimés, seulement deux opérations sont possibles :

— ajouter un comprimé dans le tube en le plaçant sur le dessus de la pile de comprimés déjà présents;

— retirer le comprimé situé sur le dessus de la pile.

Il n'est pas possible de faire d'autres opérations. Par exemple, il n'est pas possible d'inspecter un comprimé situé à l'intérieur du tube car il est inaccessible. Si l'on veut accéder à ce comprimé, pour connaître son numéro, il faut d'abord dépiler tous les comprimés situés juste au dessus de lui, puis retirer le comprimé recherché pour l'inspecter.

En programmation, des structures de données qui se comportent comme des tubes de comprimés existent. On les appelle des piles (stack en anglais). Voici comment elles sont formellement définies.

Les *piles* P sont des structures de données contenant des éléments qui peuvent apparaître en plusieurs exemplaires et possédant 2 opérations :

— `empiler(P,e)` qui ajoute l'élément e dans P;

— `depiler(P)` qui enlève et renvoie un élément de la pile P;

et qui vérifient la propriété suivante : si un élément e est ajouté avant l'élément f alors l'élément e sera enlevé après f.

Les piles sont aussi appelées *LIFO*, pour "Last In First Out", c'est à dire en français, Dernier Entré Premier Sorti.

Par exemple, dans la figure 5, on montre l'exemple d'utilisation d'une pile où l'on empile et dépile plusieurs éléments.

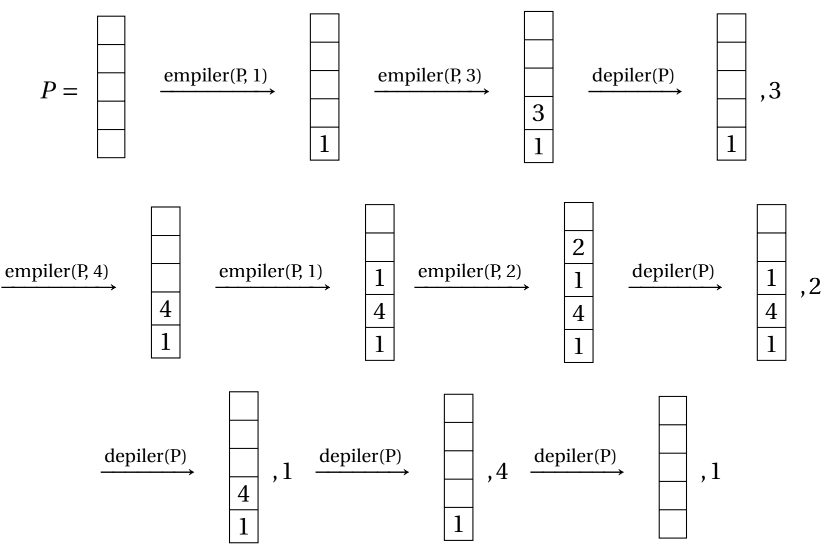


FIGURE 5 – Un exemple de manipulation d'une pile

Une API possible pour les piles est la suivante :

```
def creer_pile():
    """
```

```
    Retourne une pile vide
    """
    return NotImplemented

def empiler( p, e ):
    """
    Ajoute l'élément `e` dans la pile `p`.

    p : une pile
    e : un element
    """
    raise NotImplementedError()

def depiler( p ):
    """
    Retire le dernier élément de la pile `p` et le renvoie.

    p : une pile
    e : un element
    """
    return NotImplemented
```

Voici une implémentation possible de cette API :

```
def creer_pile( ):
    return []

def empiler( p, e ):
    p.append( e )

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

L'utilisation de la méthode `pop()` n'est pas rapide dans python. Son temps d'exécution est proportionnel au nombre d'éléments dans la liste. Ce n'est pas la cas de l'opération `append()` qui a un temps d'exécution constant quelque soit la taille de la pile (cf. la documentation `https://wiki.python.org/moin/TimeComplexity`).

En effet, en python, les listes sont des tableaux et nous verrons plus tard, que cela implique les complexités présentés ci-dessus.

Si l'on veut implémenter une pile dont toutes les opérations sont en temps constant, il faut utiliser la classe `deque` du module `collections` :

```
import collections

def creer_pile( ):
    return collections.deque()

def empiler( p, e ):
    p.append( e )

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

Généralement, les API fournissent une fonction complémentaire qui permet d'obtenir la taille de la pile, c'est à dire le nombre d'éléments contenus dans la pile.

Voici un exemple, d'API complémentaire :

```
def taille_pile( p ):
    """
    Retourne la taille de la pile passée en paramètre.
    """
    return NotImplemented
```

Cetta API s'implémente généralement ainsi :

```
def taille_pile( p ):
    return len(p)
```

Cette fonction (bien que très utile et toujours présente dans les bibliothèques) n'est pas requise pour définir une pile. Nous verrons, dans un des exercices proposés dans la feuille de TD, que l'on peut se passer de cette fonction.

5.2 Les files (FIFO)

Imaginons un distributeur de gobelets. Le distributeur de gobelets est constitué d'un cylindre opaque, ouvert en haut et en bas, avec un système en bas qui retient les gobelets pour qu'ils ne tombent pas. On remplit le distributeur par le haut et les personnes qui souhaitent un gobelet pour boire, retire les gobelets par le bas. Les gobelets sont numérotés et peuvent partager des numéros identiques. Lorsque l'on manipule le distributeur de gobelets, seulement deux opérations sont possibles :

- ajouter un gobelet dans le tube en le plaçant sur le dessus du cylindre, par dessus les gobelets déjà présents dans le distributeur;
- retirer le gobelet situé en bas du distributeur, en dessous de tous les gobelets déjà présents dans le distributeur.

Il n'est pas possible de faire d'autres opérations. Par exemple, il n'est pas possible d'inspecter un gobelet situé à l'intérieur du cylindre car il est opaque et le gobelet inaccessible. Si l'on veut accéder à ce gobelet, pour connaître son numéro, il faut d'abord retirer tous les gobelets situés juste au dessous de lui, puis retirer le gobelet recherché pour l'inspecter.

En programmation, des structures de données qui se comportent comme des distributeurs de gobelets existent. On les appelle des files (queue en anglais). On tire son nom des files d'attentes, où la première personne qui sort de la file d'attente est la première à y être entrée. Voici comment on définit formellement les files.

Les *files* F sont des structures de données contenant des éléments qui peuvent apparaître en plusieurs exemplaires et possédant 2 opérations :

- `enfiler(F,e)` qui ajoute l'élément e dans F;
- `defiler(F)` qui enlève un élément de la file F;

et qui vérifient la propriété suivante : si un élément e est ajouté avant l'élément f alors l'élément e sera enlevé avant f.

Les files sont aussi appelées *FIFO*, pour "First In First Out", c'est à dire en français, Premier Entré Premier Sorti.

Par exemple, dans la figure 6, on montre l'exemple d'utilisation d'une file où l'on enfile et défile plusieurs éléments.

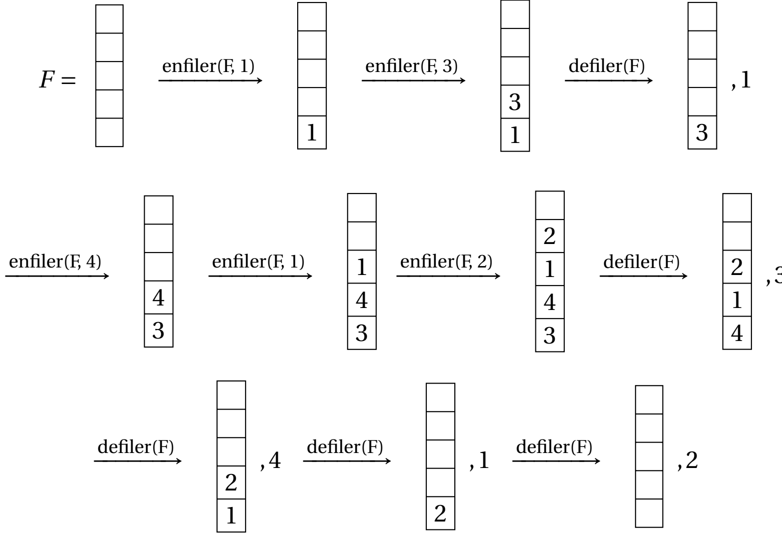


FIGURE 6 – Un exemple de manipulation d'une file

On est évidement pas obligé de décaller tous les éléments de la file à chaque fois que l'on retire un élément (comme pour le distributeur de gobelet). On obtiendrait alors la figure 7.

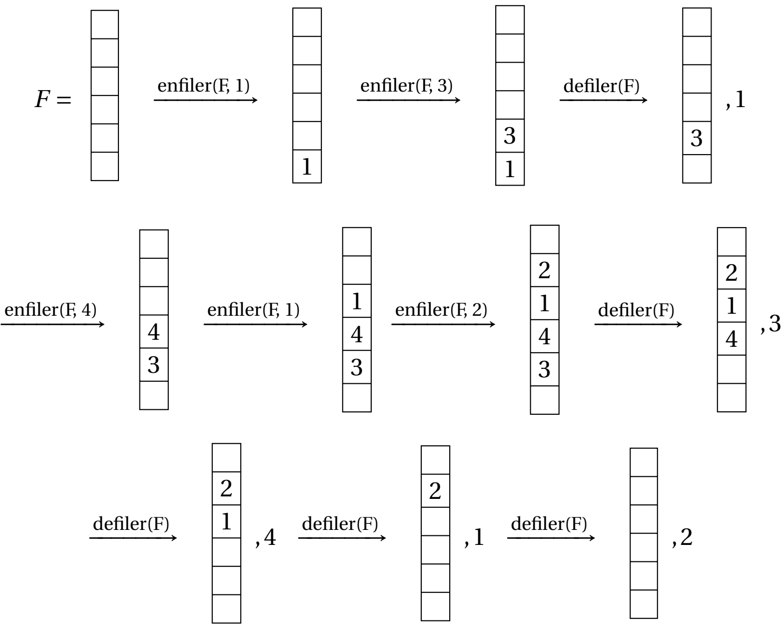


FIGURE 7 – Un exemple de manipulation en place des éléments d'une file

Une API possible pour les files est la suivante :

```
def creer_file():
    """
    Retourne une file vide
```

```
    """
    return NotImplemented

def enfiler( f, e ):
    """
    Ajoute l'élément `e` dans la file `f`.

    f : une file
    e : un element
    """
    raise NotImplementedError()

def defiler( f ):
    """
    Retire le dernier élément de la file `f` et le renvoie.

    f : une file
    e : un element
    """
    return NotImplemented
```

Voici une implémentation possible des files :

```
def creer_file( ):
    return []

def enfiler( f, e ):
    f.append( e )

def defiler( f ):
    return f.pop(0)
```

Nous avons vu dans la section précédente que l'utilisation de la méthode `pop()` n'est pas rapide en python (cf. la documentation `https://wiki.python.org/moin/TimeComplexity`).

Ainsi, si les exécutions de `creer_file()` et `enfiler(file, element)` se font en temps constants, ce n'est pas le cas de `defiler(file)` dont le temps d'exécution est proportionnel à la taille de la file.

Si on veut implémenter une file dont toutes les opérations sont garanties en temps constant, il faut utiliser le module $deque$:

```
import collections

def creer_file( ):
    return collections.deque()

def enfiler( f, e ):
    f.append( e )

def defiler( f ):
    return f.popleft()
```

Généralement, les API fournissent une fonction complémentaire qui permet d'obtenir la taille de la file, c'est à dire le nombre d'éléments contenus dans la file.

Voici un exemple, d'API complémentaire :

```
def taille_file( f ):
    """
```

```
    Retourne la taille de la file passée en paramètre.
    """
    return NotImplemented
```

Cette API s'implémente généralement ainsi :

```
def taille_file( f ):
    return len(f)
```

Cette fonction (bien que très utile et toujours présente dans les bibliothèques) n'est pas requise pour définir une file. Nous verrons, dans un des exercices proposés dans la feuille de TD, que l'on peut se passer de cette fonction.

6 La récursivité

6.1 La pile d'exécution

Dans un programme, à chaque exécution d'une fonction, le contenu des variables locales, le contenus des paramètres de la fonction, et la ligne d'exécution du code sont enregistrés dans une zone de la mémoire particulière. Cette zone de la mémoire est appelée la pile d'exécution.

La pile d'exécution est commune et unique à tout le programme.

Au début du programme, la ligne courante d'exécution ainsi que les variables du programmes sont empilés dans la pile.

Ensuite, au cours de l'exécution du programme, à chaque exécution d'une fonction f, les variables locales, la ligne courante d'exécution dans f, ainsi que les paramètres sont empilés dans la pile. L'exécution du programme continue donc à partir de la ligne courante de f qui a été initialisée à la première ligne de f. Lorsque la fonction f se termine, toutes les données empilées par f sont dépilées et perdues. Seule la valeur de retour est gardée en mémoire. Dans certains langages, cette valeur de retour est empilée dans la pile. L'exécution du programme continue alors à la ligne courante d'exécution de la fonction située en haut de la pile. Cela correspond à la dernière fonction appelée durant l'exécution du programme.

Par exemple, lorsque l'on exécute le programme suivant :

```
1   def g( a );
2       print("g(" + str(a) + ")")
3       return 1000
4
5   def h( a );
6       print("h(" + str(a) + ")")
7       return 2000
8
9   def f( a ):
10      print("f(" + str(a) + ")")
11      v = g(a+1)
12      print( v )
13      v = h(a+2)
14      print( v )
15
16  f( 1 )
17  print("fin")
```

on obtient sur le terminal, le résultat suivant :

```
f(1)
g(2)
```

```
1000
h(3)
2000
fin
```

Nous allons décrire pas à pas ce qui se passe dans la pile d'exécution.

Le programme commence à la ligne 1 et la pile d'exécution ressemble alors à :

/	ligne 1

Lorsque le programme arrive à la ligne 16, la fonction f est exécutée : un espace mémoire est créé pour la fonction f et ces données sont empilées sur la pile d'exécution. La variable locale a est initialisée à l'aide de la valeur passée en paramètre, c'est à dire 1. L'entier 1 est alors enregistré dans le petit espace mémoire nommé a associé à l'espace mémoire de f dans la pile. A ce moment, la pile est alors égal à :

f	1 (a)
	ligne 10
/	ligne 16

À ce stade, l'exécution du programme principal (symbolisé par /) est interrompu à la ligne 16 et attends, pour reprendre son activité, la fin d'exécution de toutes les fonctions empilées au dessus de lui. Reprenons le cours d'exécution du programme : la fonction f commence à s'exécuter à partir de la ligne 10.

La fonction f s'exécute jusqu'à la ligne 11. À ce moment là, son exécution est aussi interrompue pour exécuter, cette fois-ci, la fonction g. Un espace mémoire est créé pour g et cet espace est empilé sur la pile d'exécution. La pile devient alors égal à :

g	2 (a)
	ligne 2
f	? (v)
	1 (a)
	ligne 11
/	ligne 16

À ce stade, la variable v de l'espace mémoire associé à f contient une valeur inconnue. En effet, il faut attendre la valeur de retour de g pour modifier v.

La fonction g s'exécute ainsi jusqu'à ce qu'elle se termine. A la fin de son exécution, l'espace mémoire de g est défilé et perdue. Seule la valeur de retour, qui vaut 1000 est empilée sur le dessus de la pile.

On obtient la pile suivante :

f	1000 (g(2))
	? (v)
	1 (a)
	ligne 11
/	ligne 16

Cette valeur de retour est alors de suite enregistrée dans v. On obtient la pile suivante :

f	1000 (v)
	1 (a)
	ligne 11
/	ligne 16

L'exécution de f reprend donc à la ligne spécifiée dans la pile, c'est à dire à la ligne 11.
Et ainsi de suite ...
La figure 8 contient l'évolution complète de la pile d'exécution pour tout le programme.

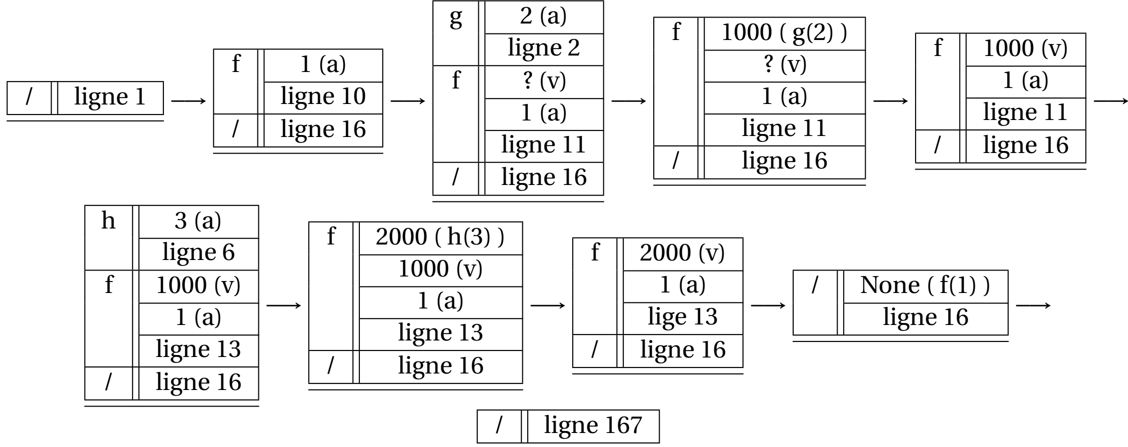


FIGURE 8 – Évolution de la pile d'exécution d'un programme

Exercice 7
Dessiner l'évolution de la pile d'exécution du programme suivant :

```
1  def prog1(var):
2      g = var - 1
3      return g
4
5  def prog2(var):
6      g = prog1(2*var)
7      r = g + prog1(var+1)
8      return r
9
10 b = 5
11 prog2(b)
```

Sous python, il est possible d'inspecter la pile avec le module `inspect`. Voici un programme python :

```
1  import inspect
2
3  def f( a, b ):
4      g(a+1, b+1)
5
6  def g( c, d ):
7      print( "La pile est : " )
8      print( inspect.stack() )
9      print( "" )
10
11     print( "Les données de la fonction située en haut de la pile sont : " )
12     print( inspect.stack()[0] )
13     print( "" )
```

```
14
15      print( "Les variables de la fonction située en haut de pile sont : " )
16      print( inspect.getargvalues( inspect.stack()[0][0] ) )
17
18  f( 1, 2 )
```

A l'exécution de ce programme, on obtient le résultat suivant :

```
1   La pile est :
2   [
3     (<frame object at 0x7f59363a6938>, 'prog.py', 8, 'g', [' print( inspect.stack() )\n'], 0),
4     (<frame object at 0x7f59363a93f0>, 'ptog.py', 4, 'f', [' g(a+1, b+1)\n'], 0),
5     (<frame object at 0x7f59364f6050>, 'porg.py', 18, '<module>', ['f( 1, 2 )\n'], 0)
6   ]
7
8   Les données de la fonction située en haut de la pile sont :
9   (<frame object at 0x7f59363a6938>, 'prog.py', 12, 'g', [' print( inspect.stack()[0] )\n'], 0)
10
11  Les variables de la fonction située en haut de pile sont :
12  ArgInfo(args=['c', 'd'], varargs=None, keywords=None, locals={'c': 2, 'd': 3})
```

Exercice 8

1. En analysant le texte écrit sur le terminal, repérer la ligne d'exécution courante du programme principal, de f et de g.
2. Sans regarder le code, quels sont les variables locales de g et leurs valeurs au moment où le texte a été affiché sur le terminal ?

Pour finir, rappelez-vous que les listes, les chaînes de caractères et les instances d'objet, ne sont pas stockés dans la pile, mais ailleurs dans la mémoire. Ce qui est stocké dans la pile, ce sont les variables, c'est à dire des petits espaces mémoires qui contiennent des références vers ces gros objets. Ainsi, quand une fonction termine son exécution, si les variables de son espace mémoire sont dépilés et détruites, ce n'est pas le cas de ces gros objets créés qui subsistent tant qu'une variable ou une structure de données garde une référence vers eux.

Exercice 9

Qu'affiche le programme suivant ? Dessiner l'évolution de sa pile d'exécution ainsi que de la mémoire du programme.

```
1   def prog1(v):
2     v[2] = [5,6]
3     return v[0]
4
5   def prog2(var):
6     var[0] = [4,5]
7     g = prog1(var)
8     var[1] = g
9
10  li = [1,2,3]
11  prog2(li)
12  print(li)
```

Les langages de programmation qui utilisent des fonctions utilisent généralement une pile d'exécution pour organiser la mémoire associée à l'exécution de ses fonctions. C'est le cas, par exemple, du

langage C. La structure de la mémoire d'un langage en C est donnée à la figure 9.

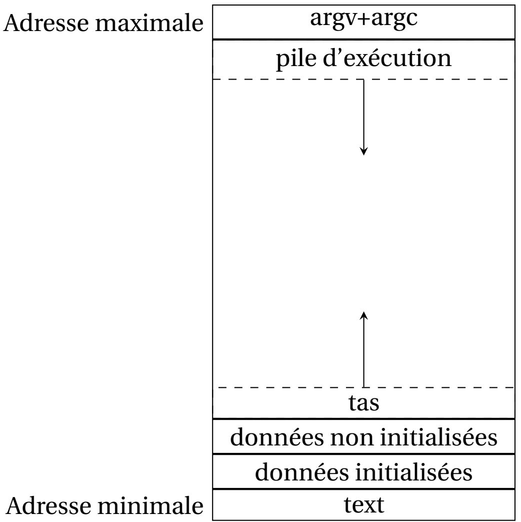


FIGURE 9 – Gestion de la mémoire en C.

On retrouve dans cette figure la pile d'exécution, mais aussi le tas. Le tas c'est la zone mémoire où les gros objets (listes, certains types de chaînes de caractères, etc...) sont stockés en mémoires.

6.2 La récursivité

On dit qu'une fonction f est récursive, si, durant l'exécution de f, la fonction f est de nouveau appelée.

Autrement dit, la fonction f est récursive, si au cours de l'exécution du programme, la fonction f apparaît au moins 2 fois en même temps dans la pile d'exécution.

Voici un exemple de programme récursif qui permet de calculer factoriel(n) $= 1 \times 2 \times \cdots \times n$. Par exemple, factoriel(4) $= 1 \times 2 \times 3 \times 4$.

```
1 def factoriel( n ):
2     if n == 0:
3         return 1
4     return factoriel( n-1 )*n
5 
6 print( factoriel( 4 ) )
```

La pile d'exécution du programme précédent est représentée à la figure 10 :

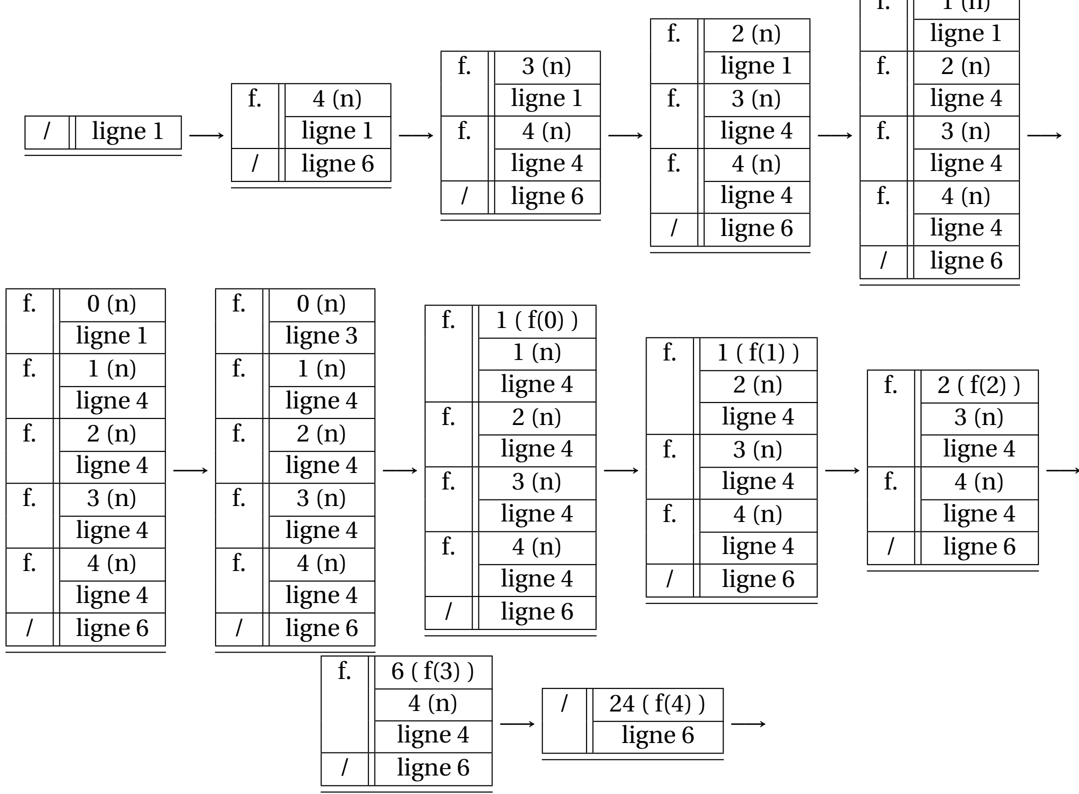


FIGURE 10 – Pile d'exécution de factoriel(4)

Exercice 10

1. Pourquoi le programme précédent est récursif?
2. Analyser l'exécution du programme précédent à l'aide de l'évolution de sa pile d'exécution et expliquez pourquoi le programme s'arrête.
3. Expliquez pourquoi ce programme calcul correctement la factorielle de n.
4. Modifiez une seule valeur du programme pour que le programme fasse une boucle infinie (ne s'arrête pas). Justifiez pourquoi le programme ne s'arrêtera pas.

Pour qu'un programme f récursif s'arrête, il faut que :

1. une portion de code, ne contenant pas d'appel à f, et terminant l'exécution de f, s'exécute uniquement pour certaines valeurs de paramètres particuliers. On appelle condition d'arrêt, la condition logique qui active l'exécution de cette portion de code. Dans l'exemple précédent, la valeur d'arrêt est 0 et la condition d'arrêt est $n == 0$. La portion de code arrêtant f est située dans les lignes 1 et 2.
2. A chaque fois que f appelle f, les valeurs passées en paramètres changent et se rapprochent des valeurs qui activent la condition d'arrêt. Dans l'exemple précédent, à la ligne 4, factoriel(n) appelle factoriel($n-1$). Les paramètres changent donc à chaque fois et la valeur passée en paramètre se rapproche toujours de la valeur qui active la condition d'arrêt, à savoir 0 .

Tout programme récursif doit donc impérativement contenir un code d'arrêt avec une condition d'arrêt clairement identifiée et facilement identifiable. De plus, l'appel récursif doit clairement montrer, à chaque appel, que les valeurs sont modifiés pour se rapprocher de la valeur d'arrêt.

7 Recherche d'un élément dans une tableau

Dans cette section nous allons nous intéresser à la recherche d'un élément dans un tableau. Pour cela, nous allons restreindre le problème aux tableaux d'entiers, et nous allons chercher à savoir si un entier e donné se trouve dans un tableau t contenant uniquement des entiers. Le tableau t peut éventuellement contenir plusieurs fois le même entier.

7.1 Recherche linéaire

En toute généralité, si le tableau t n'est pas ordonné, il est nécessaire de passer en revu tous les éléments du tableau pour savoir si l'entier e se trouve dans t. On réalise cela à l'aide du programme suivant, qui renvoie l'index du premier élément e dans t si e est dans t et `None` sinon.

```
def recherche_lineaire( t, e ):
    for i in range( len(t) ):
        if t[i] == e :
            return i
    return None
```

Le temps d'exécution de ce programme peut être évalué en comptant le nombre d'instructions réalisées. Si l'on considère que modifier la valeur de i dans la boucle `for` compte aussi pour 1 instruction, et que les opérations usuelles =, ==, +, -, *, etc ... comptent pour une seule opération, alors l'algorithme précédent a réalisé, dans le pire des cas : $2.n$ opérations, où n est le nombre d'éléments du tableau t.

7.2 Recherche dichotomique

Il est possible d'améliorer la recherche d'un élément d'un tableau si le tableau est trié par ordre croissant. On utilise, pour cela, une recherche dichotomique.

L'idée de la recherche dichotomique, consiste à comparer l'entier e à chercher avec l'entier situé au milieu du tableau. Si e est plus petit ou égal, alors on va chercher l'entier uniquement dans la partie gauche du tableau. Si e est strictement plus grand, alors on va chercher l'entier uniquement dans la partie droite. On s'arrête lorsqu'il ne reste plus qu'un seul élément à inspecter. Si ce dernier élément n'est pas égal à e, alors c'est que le tableau t ne contient pas e.

Cela nous donne ainsi l'algorithme suivant :

```
def recherche_dichotomique( t, e ):
    if len(t) == 0 :
        return None
    i = 0
    j = len( t ) – 1
    while( i != j ):
        milieu = (i+j)//2
        if t[milieu] < e:
            i = milieu + 1
        else :
            j = milieu
    if( t[i] == e ):
        return i
    else:
        return None
```

Exercice 11
Proposer une version récursive de l'algorithme de recherche dichotomique.

7.3 Estimer le nombre d'instructions de l'algorithme de recherche dichotomique (optionnel-à regarder à la fin de la séance)

Nous allons maintenant estimer le nombre d'instructions $C(t)$ nécessaires à cet algorithme pour chercher un élément dans un tableau t de taille n.

Si le tableau n'est pas vide, alors le programme réalise 5 opérations en dehors de la boucle. De même, le programme réalise entre 6 et 7 opérations à chaque tour de boucle. Ainsi, si le tableau n'est pas vide, le nombre d'instructions, réalisées par le programme, peut être encadré par

$$5 + 6.b(t) \leq C(t) \leq 5 + 7.b(t)$$

où $b(t)$ est le nombre de fois que l'on a exécuté la boucle `while` de l'algorithme.

Soit a le plus grand entier tel que $2^{a-1} < n \leq 2^a$, où n est la taille du tableau. Soit u_k le nombre d'éléments situés entre i et j inclus à la fin de l'exécution de la $k^{\text{ème}}$ boucle. On a alors

$$2^{a-k-1} < u_k \leq 2^{a-k}. \tag{1}$$

En effet, il suffit de faire une récurrence sur k pour le démontrer.

Le programme s'arrête au moment où $i = j$, c'est à dire lorsque $u_k = 1$. Cette condition se réalise quand $k = a$ (voir Équation 1). Comme $2^{a-1} < n \leq 2^a$ On en déduit que $b(t) = a = \left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil$.

L'algorithme réalise donc entre $5 + 6.\left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil$ et $5 + 7.\left\lceil \frac{\ln(n)}{\ln(2)} \right\rceil$ opérations.

Lorsque n est grand, ce nouvel algorithme est bien plus efficace que la recherche linéaire. Ceci peut être observé à l'aide de la figure 11.

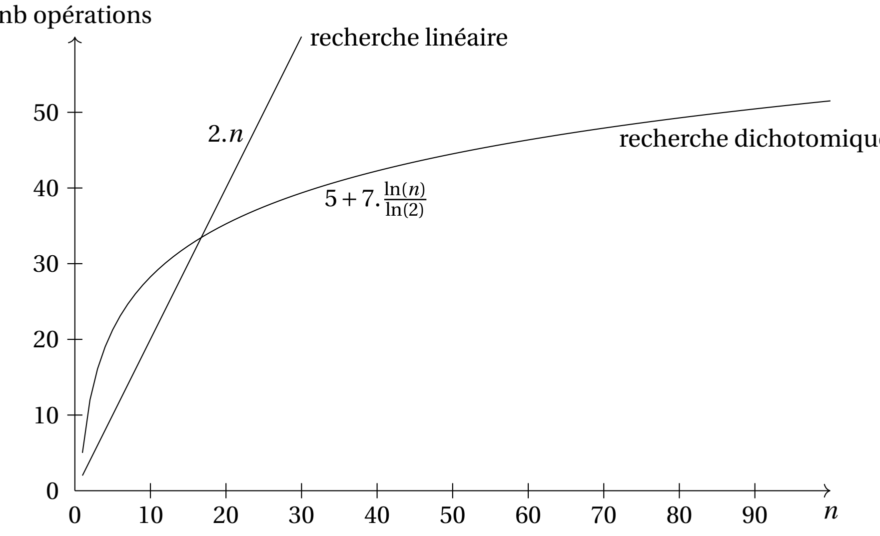


FIGURE 11 – Comparaison du nombre d'opérations réalisées par les différents algorithmes de recherche (n est la taille du tableau).

8 Listes

Les *listes* sont des structures de données qui contiennent une séquence d'éléments dont les éléments peuvent apparaître en plusieurs exemplaires. Chaque élément est encapsulé dans une structure de données appelée *cellule* qui permet de récupérer l'élément, et qui permet d'accéder à la structure encapsulant l'élément suivant dans la séquence. On parle alors de *liste chaînée*.

8.1 Listes chaînées ou listes simplement chaînées

Une *liste chaînée* (ou une *liste simplement chaînée*) est une structure de données l encodant une séquence d'éléments $(l_1, l_2, \dots, l_n)$, où les éléments peuvent apparaître avec répétition.

Dans une liste chaînée, chaque élément l_i est encapsulé dans une cellule c_i.

Chaque cellule c_i contient deux informations :

— la première, appelée *valeur*, est l'élément que contient la cellule; cette valeur vaut l_i ;

— la deuxième, appelée *successeur*, est une référence vers la cellule suivante c_{i+1} si $i \le n-1$ et la référence $None$ sinon.

Par exemple, dans la figure 12, la cellule c_2 contient la valeur l_2 et la référence à la prochaine cellule qui est c_3.

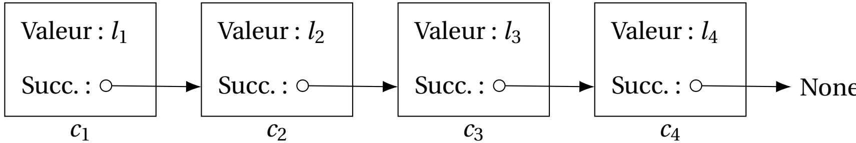


FIGURE 12 – Un liste simplement chaînée

Les cellules $c_1, c_2, \dots, c_n$ sont toutes deux à deux distincts même si deux cellules peuvent encapsuler deux éléments identiques. Par exmple, dans la figure 13, la liste simplement chaînée est $[2,1,4,2]$, qui contient 4 cellules c_1, c_2, c_3 et c_4 distinctes, mais deux valeurs identiques, celle de c_1 et celle de c_4.

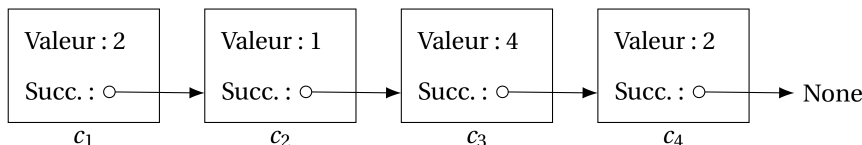


FIGURE 13 – La liste chaînée $[2,1,4,2]$.

On manipule les listes à l'aide de la référence à sa première cellule. Ainsi, si la référence à la première cellule est None, on dit que la liste est vide.

On peut implémenter les listes chaînées à l'aide d'une structure de données l munie de l'API suivante :

```
def create_list():
  """
  Créer et retourne une liste vide.
  """
  return NotImplemented

def create_cell(value, successeur):
  """
  Créer et retourne une cellule contenant la valeur
  `valeur` et le successeur `successeur`.
  """
  return NotImplemented

def change_cell(cell, valeur, successeur):
  """
  Change les attributs `valeur` et `successeur` de la cellule
  `cell` passée en paramètre.
  """
```

```
    raise NotImplementedError()

def push_front(l, e):
    """
    Ajoute l'élément `e` au début de la liste `l`. Cette fonction créé une nouvelle cellule et l'insère en début
    de liste.
    """
    return NotImplemented

def remove_front(l):
    """
    Retire le premier élément de la liste si il existe.
    """
    return NotImplemented

def first_cell( l ):
    """
    Retourne la première cellule de la liste `l` et None si la liste est vide.
    """
    return NotImplemented

def next_cell( cell ):
    """
    Retourne le successeur de la cellule `cell`.
    """
    return NotImplemented

def value_cell( cell ):
    """
    retourne la valeur de la cellule `cell`.
    """
    return NotImplemented
```

En python, on peut l'implémenter de la manière suivante :

```
def create_list():
        return {'first':None}

def create_cell(value, successeur):
    return {'value': value, 'next': successeur}

def push_front(l , e):
        l['first'] = create_cell(e, l['first'])

def remove_front(l):
    if not l['first'] is None :
        l['first'] = l['first']['next']

def first_cell(l):
    return l['first']

def next_cell(cell):
    return cell['next']
```

```
def value_cell(cell):
    return cell['value']
```

Cette impélmentation s'utilise alors ainsi :

```
l = create_list()
push_front(l, 3)
push_front(l, 5)
push_front(l, 2)

it = first_cell(l)
while( it != None ):
    print( value_cell(it) )
    it = next_cell(it)

remove_front(l)
remove_front(l)
remove_front(l)
```

Exercice 12
Testez le programme précédent et analysez la mémoire à l'aide de PythonTutor .

8.2 Listes doublement chaînées

Une *liste doublement chaînée* est une liste chaînée dont les cellules c_i contiennent une information supplémentaire appelée *prédécesseur* qui vaut None si $i = 1$ et qui est une référence vers la cellule c_{i-1} lorsque $2 \leq i \leq n$.

Par exemple, la figure 14 montre la liste doublement chaînée associée à $[1,2,4,2]$.

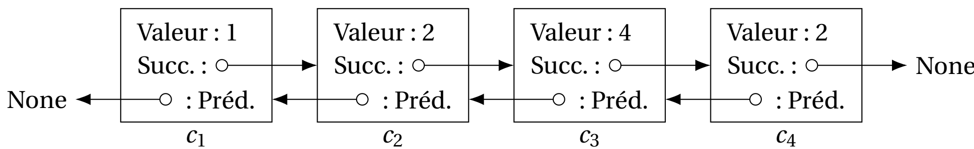


FIGURE 14 – La liste doublement chaînée $[1,2,4,2]$.

L'API des listes doublement chaînées contient l'API des listes simplement chaînées à laquelle on ajoute les fonctions suivantes :

```
def push_back(l, e):
    """
    Ajoute l'élément `e` àla fin de la liste `l`.
    """
    return NotImplemented

def last_cell(l):
    """
    Renvoie la dernière cellule de la liste `l`.
    """
    return NotImplemented
```

```python
def prev_cell( cell ):
    """
    Retourne la cellule prédécesseur àla cellule `cell`.
    """
    raise NotImplementedError()
```

Exercice 13
En vous inspirant du code des listes simplement chaînées, implémentez les listes doublement chaînées.

Les listes doublement chaînées sont pratiques à utiliser car il est possible de retirer n'importe quelle cellule de la liste et de la replacer ensuite, comme le montre la figure 15 avec la cellule c_2.

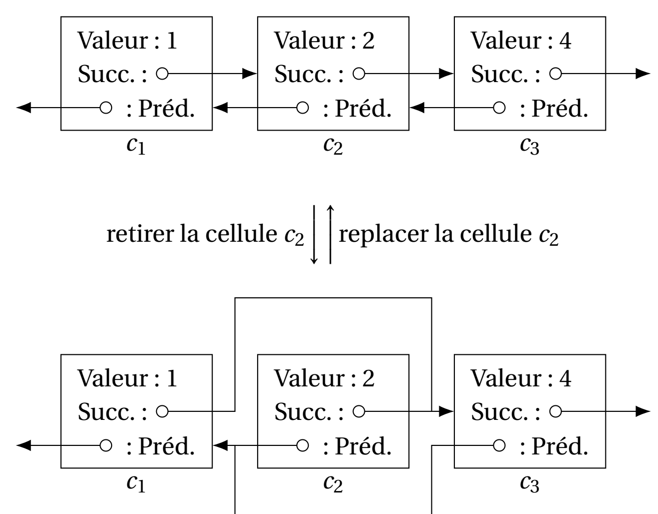


FIGURE 15 – Retirer et replacer une cellule d'une liste doublement chaînée.

Cependant, avec l'implémentation proposée, la structure interne de la liste peut être détruite lorsque l'on décide de retirer la première (resp. la dernière) cellule de la liste. En effet, la structure de données de la liste qui contient la référence à la première et à la dernière cellule n'est pas mis à jour.

La solution à ce problème consiste à créer une liste doublement chaînée circulaire contenant une cellule dite *cellule de fin de liste*, notée c_{fin}. Ainsi, la structure de données l de la liste contient juste la référence à la cellule de fin de liste. Ensuite, la première cellule de la liste est le successeur de la cellule de fin de liste. Enfin, la dernière cellule de la liste doublement chaînée est la cellule prédécesseur de la cellule de fin de liste.

La figure 16 montre un exemple de liste doublement chaînée utilisant une cellule de fin de liste.

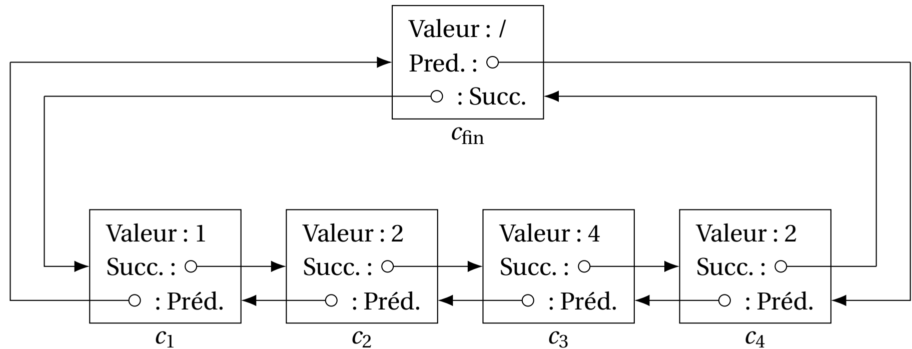


FIGURE 16 – La liste doublement chaînée 1, 2, 4, 2] avec une cellule de fin de liste.

Lorsque l'on implémente des listes doublement chaînées avec une cellule de fin de liste, alors il est possible d'ajouter à l'API, en toute sécurité, les fonctions suivantes :

```
def end_cell(l):
    """
    Retourne la cellule de fin de liste de la liste `l`.
    """
    return NotImplemented

def remove_cell( cell ):
    """
    Retire la cellule de la liste.
    """
    raise NotImplementedError()

def replace_cell( cell ):
    """
    Annulle l'action de remove_cell : replace la cellule dans la liste.
    """
    raise NotImplementedError()

def insert_cell_before( cell1, cell2 ):
    """
    Insère la cellule `cell1` juste avant la cellule `cell2` dans la liste.
    """
    raise NotImplementedError()

def insert_cell_after( cell1, cell2 ):
    """
    Insère la cellule `cell1` juste après la cellule `cell2` dans la liste.
    """
    raise NotImplementedError()
```

Exercice 14
Implémentez les listes doublement chaînées en utilisant la technique de la cellule de fin de liste.

9 La Complexité

9.1 Présentation du problème

On cherche à évaluer le temps et l'espace mémoire que va utiliser un programme.

La complexité en temps d'exécution (resp. en mémoire) d'un programme, c'est une formule mathématique qui prends en paramètre les données initiales du programme et qui évalue le temps exécuté par le programme (resp. la taille de la mémoire utilisée par le programme).

Dans la pratique il est difficile de déterminer le temps d'exécution d'un programme car il est difficile de connaître le temps de chaque opération du langage. Les opération du langages sont : +, ∗, −, /, ==, %, $T[i]$, $return$, =, etc...

Nous allons donc supposé que chaque opération s'exécute en temps constant et nous allons compter à la place le nombre d'opération réalisés par un programme.

Soit fct un programme, on notera par $\mathscr{C}(fct)$ la complexité de fct.

Par exemple, le programme suivant,

```
def fct( n ): # initialisation de n : 1 opération
    r = 0 # 1 opération
    for i in range(n): # n*( incrément + corps de boucle ) = n*( 1 + 2 )
        r = r + 1 # | 2 operations
    return r # 1 opération
```

réalise $3+3n$ opération et a donc pour complexité : $\mathscr{C}(fct) = 3+3n$.

Analysons l'exemple suivant :

```
def max( T ): # init. : 1 opération
    res = T[0] # 1 opération
    for i in range( len(T) ): # C1
        if res < T[i] : # | C2
            res = T[i] # | | 2 operations
    return res # 1 opération
```

Soit n la taille du tableau T. La complexité $C2$ du saut conditionnel `if` est de 3 opérations pour la condition auquel il faut ajouter les opérations du corps du `if`. Le corps du `if` ajoute 2 ou 0 opérations supplémentaires selon que la condition est vraie ou fausse. On obtient que $3 \leq C2 \leq 5$. La complexité $C1$ de la boucle `for` est de $n \times$ (incrément + corps de boucle). Ainsi,

$$C1 = \begin{cases} n(1+3) & \text{si la condition est fausse;} \\ n(1+5) & \text{si la condition est vraie.} \end{cases}$$

La complexité du programme est donc :

$$3+4n \leq \mathscr{C}(max) \leq 3+6n.$$

La borne inférieure est atteinte si le plus grand élément du tableau T se trouve en première position et la borne supérieure est atteinte si le tableau T contient des éléments tous distincts triés par ordre croissant.

Voici une autre implémentation (horrible et idiote) du même programme :

```
def max2( T ): # init. : 1 opération -- on pose n= len(T)
    T = list( T ) # = + Copie de tableau : 1+n opérations
    for i in range( len(T) ): # C1
        for j in range( len(T) ): # | C2
            if T[i] < T[j]: # | | C3
                T[i] = T[j] # | | | 3 opérations
    return T[ len(T)-1 ] # 3 opérations
```

Par un raisonnement analogue au programme précédent, on a

$$\begin{aligned} 3 \leq C3 \leq 6, \\ n(1+3) \leq C2 \leq n(1+6), \\ n(1+4n) \leq C1 \leq n(1+7n). \end{aligned}$$

Ce qui donne la complexité suivante :

$$5 + 2n + 4n^2 \leq \mathscr{C}(max2) \leq 5 + 2n + 7n^2.$$

9.2 La complexité en pire cas et en meilleur cas

Le but de l'algorithmique est de comparer ensemble différents algorithmes qui réalisent la même tâche.

Pour pouvoir comparer ces algorithmes, on compare le nombre d'opérations réalisées par les différents algorithmes en fonction de entrées que ces algorithmes peuvent traiter.

Pour faire cela, on commence par mesurer la taille des paramètres d'entrées en proposant une fonction qui prends en paramètre les paramètres de la fonction et qui renvoie un entier mesurant cette taille.

Par exemple, pour une fonction travaillant sur les liste, la taille qui mesurera les entrées de la fonction pourra être la taille de la liste.

Ensuite, pour chaque programme, on cherche, pour une taille donnée, les paramètres d'entrées qui nécessite le plus d'opérations de calcul. On compte ensuite ce nombre d'opérations, puis on détermine la fonction qui à une taille donnée, détermine le temps d'exécution (en nombre d'opération) dans le pire cas. cette fonction appelée *complexité dans le pire cas* de notre programme.

Pour déterminer quel programme est le plus adapté, on compare alors ces fonctions entre elles.

De la même manière on appelle *complexité dans le meilleur cas*, la fonction qui, à une taille donnée, détermine le temps d'exécution pour l'ensemble des paramètres le plus favorable.

Par exemple, étudions les deux programmes suivants qui prends en paramètre un tableau codé sous forme de liste chaînée, pour le premier programme et sous forme de tableau (type `list` en python) et déterminons le temps nécessaire au programme pour déterminer le nombre d'éléments contenus par le tableau.

```
def nb_element_1(l);
    res = 0
    cell = first(l)
    while( not cell is end(l) ):
    cell = next( l, cell )
    res += 1
  return res
```

```
def nb_element_2(l);
  return len(l)
```

Pour comparer ces deux fonctions, il faut définir une fonction $taille$ qui nous servira à mesurer la taille des entrées. Dans notre cas, on utilisera le nombre d'éléments du tableau.

Pour chaque taille on détermine une jeu d'entrées qui maximise le temps de calcul. Ici, pour une taille n donnée, n'importe quel tableau comprenant n éléments maximise le temps d'exécution des deux programmes. On pourra prendre un tableau arbitraire pour déterminer la complexité des deux algorithmes.

Enfin, on détermine les fonctions f_1 et f_2 qui déterminent les complexités maximales de respectivement $nb_element_1$ et $nb_element_2$:

$$f_1(n) = 4 + n \times 7$$

$$f_2(n) = 2$$

On se rends compte alors que quelque soit la taille de donnée n, le deuxième algorithme est plus rapide que le premier.

9.3 Comparer les algorithmes en utilisant des comparaisons asymptotiques

Nous avons vu comment comparer deux algorithmes ensemble, en comparant leurs complexités.

Le calcul des complexités n'est pas toujours aisée.

En fait, connaître la valeur exacte du nombre d'opérations n'est pas une information pertinente : savoir qu'un programme réalise $n+4$ opérations, $n+2$ opérations ou bien $2\times n+1$ n'apporte pas beaucoup d'informations utiles. En fait, ce qui importe c'est que le programme réalise a peu près $\alpha \times n$ opérations quand la taille de donné n devient grande.

En effet, les constantes 1, 4 et 2 deviennent négligeables devant la taille de donnée n grandissante. De même, il n'est pas utilise de connaître la constante α quand on compare un programme dont la complexité est $\alpha \times n$ à un programme de complexité $\beta \times n^2$. En effet, quand n grandira, le premier programme fera toujours moins d'opérations que le second.

Ainsi, quand on détermine la complexité d'un programme on réalise des calculs de complexité en ignorants les constantes et en ne gardant que le terme dominant, c'est à dire la fonction mathématique en n la plus grande dans une expression.

En mathématique on appelle cela le calcul asymptotique et des outils ont été créé pour réaliser avec rigueur ce calcul et pour déterminer les fonctions mathématiques les plus grande dans une expression. Il s'agit du O qui est définit comme suit.

Soit f et g deux fonctions (dits de complexité) de $\mathbb{N}$ dans $\mathbb{R}^+$. On dit que g domine f, ou que f est un grand O de g si

$$\exists \alpha \in \mathbb{R}^+, \exists N \in \mathbb{N}^* \text{ tq.} \forall n \in \mathbb{N}, \text{ si } n \geq N \text{ alors } f(n) \leq \alpha \cdot g(n).$$

On note cette dernière relation par $f = \mathcal{O}(g)$.

Cette dernière phrase mathématique se lit ainsi : "Il existe un réel positif non nul α, il existe un entier naturel N non nul, tel que pour tout entier n, si n est plus grand que N alors $f(n)$ est plus petit que α fois $g(n)$."

On peut interpréter cette phrase avec un point de vue informaticien de la façon suivante :

— la fonction f est la complexité réelle du programme et g est son asymptotique.

— il existe une constante de temps α, il existe une taille de donnée minimale N telle que lorsque les paramètres en entrée dépassent la taille N, alors la complexité du programme $f(n)$ est majorée par α fois $g(n)$.

9.4 Règles de calculs pour $\mathcal{O}$

Il existe de nombreuses règles de calculs qui se démontrent à partir de la définition précédente. Voici une liste non exhaustive :

— $\mathcal{O}(f+g) = \mathcal{O}(f) + \mathcal{O}(G)$;

— $\mathcal{O}(f) + \mathcal{O}(G) = \mathcal{O}(g+g)$;

— $\mathcal{O}(\alpha \times f) = O(f)$;

— $\mathcal{O}(f) = O(\alpha \times f)$;

— $\mathcal{O}(f) \times \mathcal{O}(g) = \mathcal{O}(f \times G)$;

— $\mathcal{O}(f \times G) = \mathcal{O}(f) \times \mathcal{O}(g)$;

— $\mathcal{O}(n^i) = \mathcal{O}(n^j)$ si $i \leq j$;

— $\mathcal{O}(n^j) \neq \mathcal{O}(n^i)$ si $i \leq j$;

— $\mathcal{O}(\ln(n)) = \mathcal{O}(n)$

— $\mathcal{O}(n) \neq \mathcal{O}(\ln(n))$

— $\mathcal{O}(n^i) = \mathcal{O}(n!)$;

— $\mathcal{O}(n!) \neq \mathcal{O}(n^i)$.

Pour déterminer, dans un programme, les complexités de vos fonctions, vous pouvez utiliser les formules présentés dans l'annexe B.

9.5 Quelques exemples

9.6 Complexité en moyenne et complexité amortie

10 Algorithmes de Tri

Soit T un tableau non ordonné d'entiers, pouvant contenir plusieurs fois le même entier.

Nous allons nous intéresser aux algorithmes qui permettent de trier ce tableau.

10.1 Tri à bulles

L'algorithme de tri à bulles consiste à répéter, jusqu'à ce que le tableau soit trié, une étape de tri qui consiste à parcourir les éléments du tableaux et à échanger deux éléments successifs s'ils ne sont pas dans l'ordre croissant.

Le programme Python de tri est le suivant :

```
def echanger(t, i, j):
    tmp = t[i]
    t[i] = t[j]
    t[j] = tmp

def etape_tri_bulle( t ):
    est_trie = True
    for i in range( len(t)-1 ):
        if( t[i]>t[i+1] ):
            echanger( t, i, i+1 )
            est_trie = False
    return est_trie

def tri_bulle( t ):
    est_trie = False
    while not( est_trie ):
        est_trie = etape_tri_bulle( t )
```

Si le tableau t est déjà trié, cet algorithme peut se terminer très rapidement, en réalisant juste une itération de `etape_tri_bulle(t)`.

Cependant, les performances chutent lorsque le tableau est quelconque. On peut évaluer le nombre d'opérations effectuées par cet algorithme, dans le pire cas. Soit n le nombre d'éléments contenus dans un tableau. Commençons par évaluer le nombre d'opérations effectuées par `etape_tri_bulle` : cette fonction réalise 1 opération en dehors de la boucle `for` et entre 3 et 8 opérations par boucle à l'intérieur de la boucle `for`. Le programme `etape_tri_bulle` réalise donc entre $1+3(n-1)$ et $1+8(n-1)$ opérations. Étudions maintenant la complexité de `tri_bulle`. À la fin du $k^{\text{ème}}$ appel de `etape_tri_bulle(t)`, le tableau t contient les k plus grand éléments de t dans les k dernières cases, triées par ordre croissant. Ainsi, à la $n-1^{\text{ème}}$ exécution de `etape_tri_bulle(t)` les $n-1$ plus grands éléments sont triés et situés à la fin, donc le tableau est complètement trié. On en déduit que `etape_tri_bulle(t)` est exécuté, dans le pire cas, $n-1$ fois. Ce pire cas est atteint pour le tableau $[n, n-1, n-2, \ldots, 1]$. Ainsi, dans le pire cas, l'algorithme réalise entre $1+(n-1).(1+1+3.(n-1)) = 3.n^2-5.n+3$ et $1+(n-1).(1+1+8(n-1)) = 8.n^2-14.n+7$ opérations.

Ce nombre d'opération est un polynôme de degrés 2 en n. On dit que cet algorithme est de complexité quadratique dans le pire cas.

Dans le meilleur cas, si le tableau est déjà trié, cet algorithme réalise $1+3(n-1)$ opérations qui est un polynôme de degrés 1 en n. On dit alors que ce programme est de complexité linéaire dans le meilleur des cas.

Quand on parle de complexité d'un programme, on parle généralement de sa complexité dans le pire des cas. Ainsi, l'algorithme de tri à bulle est de complexité quadratique.

10.2 Tri par insertion (tri du joueur de carte)

Le tri par insertion, appelé aussi tri du joueur de carte, est le tri réalisé généralement par les joueurs de cartes. Habituellement, les joueurs de cartes trient ses cartes au fur et à mesure qu'ils les obtiennent en insérant la nouvelle carte à la bonne position parmi les cartes déjà triées. Cet algorithme s'écrit en Python de la façon suivante :

```
def echanger(t, i, j):
    tmp = t[i]
    t[i] = t[j]
    t[j] = tmp

def inserer( t, i ):
    j = i
    while j > 0 and t[j] < t[j–1] :
        echanger( t, j, j–1 )
        j = j – 1

def tri_insertion( t ):
    for i in range( len(t) ):
        inserer( t, i )
```

Nous allons étudier la complexité dans le pire cas de `tri_insertion`. Soit $c(t,i)$ le nombre d'opérations réalisées par `inserer(t,i)`. Dans le pire cas, la boucle de `inserer(t,i)` est exécuté i fois. Ainsi, dans le pire cas, $c(t,i) = 1+10i$. Lors de l'exécution de `tri_insertion(t)`, ce pire cas est atteint pour toutes les exécutions de `inserer(t,i)` lorsque le tableau t initial vaut [n, n-1, n-2, ..., 3, 2, 1].

Ainsi, dans le pire cas, le programme `tri_insertion(t)` réalise :

$$\sum_{i=0}^{n-1}(1+c(t,i)) = \sum_{i=0}^{n-1}(1+1+10i) = 2n+10\frac{n(n-1)}{2} = 5n^2-3n$$

opérations (voir propriété 2 pour le détail des calculs).

Cet algorithme est donc un algorithme quadratique.

10.3 Tri par sélection

Le tri par sélection est un tri qui consiste à itérer n étapes où la $k^{\text{ème}}$ étape consiste à chercher le $k^{\text{ème}}$ plus petit élément et à l'échanger avec l'élément situé en position k dans le tableau.

On obtient ainsi l'algorithme suivant :

```
def echanger(t, i, j):
    tmp = t[i]
    t[i] = t[j]
    t[j] = tmp

def recherche_min( t, i ):
    position_min = i
```

```
    for k in range( i+1, len(t) ):
        if( t[ k ] < t[ position_min ] ):
            position_min = k
    return position_min

def tri_insertion( t ):
    for i in range( len(t) ):
        echanger( t, i, recherche_min( t, i ) )
```

Si l'on note par $c(t,i)$ la complexité de `recherche_min(t,i)`, alors on obtient l'encadrement $2+2(n-i-1) \leq c(t,i) \leq 2+3(n-i-1)$. Le nombre d'opérations réalisées par le programme `tri_insertion` est donc compris antre $\sum_{i=0}^{n-1}(6+2(n-i-1)) = 6+2\frac{n(n-1)}{2}$ et $\sum_{i=0}^{n-1}(6n+3(n-i-1)) = 6n+3\frac{n(n-1)}{2}$.

L'algorithme de tri par sélection est donc quadratique.

10.4 Tri rapide

Le principe du tri rapide consiste à choisir un élément p du tableau, appelé pivot, puis à trier le tableau en mettant les éléments plus petits que p à gauche de p et les éléments plus grands que p à droites de p, puis a recommencer le processus, jusqu'à ce que le tableau soit entièrement trié, sur le tableau de gauche d'une part et sur le tableau de droite d'autre part.

On obtient ainsi l'algorithme suivant :

```
def echanger(t, i, j):
    tmp = t[i]
    t[i] = t[j]
    t[j] = tmp

def partitioner( t, min, max ):
    pivot = max
  i = min
    j = max − 1
    while( i<j ):
        if t[i] < t[pivot] :
            i = i + 1
        else:
            echanger( t, i, j )
            j = j − 1
    if t[i] < t[pivot] :
        echanger( t, i+1, pivot )
        pivot = i+1
    else :
        echanger( t, i, pivot )
        pivot = i
    return pivot

def tri_rapide_recursif( t, i, j ):
    if( i < j):
        pivot = partitioner( t, i, j )
        tri_rapide_recursif( t, i, pivot−1 )
        tri_rapide_recursif( t, pivot+1, j )

def tri_rapide( t ):
    tri_rapide_recursif( t, 0, len(t)−1 )
```

La complexité de ce programme est quadratique dans le pire cas. Cet algorithme est tout de même très utilisé car il est rapide dans beaucoup de cas usuels.

10.5 Tri fusion

Le tri fusion consiste à couper le tableau en 2, à trier le tableau de gauche à l'aider d'un autre tri fusion, à trier le tableau de droite avec le même algorithme, puis à fusionner les deux tableaux.

On obtient ainsi l'algorithme suivant :

```
def fusion( t, min, milieu, max ):
    tmp = []
    i = min
    j = milieu + 1
    while i <= milieu or j <= max :
        if i > milieu :
            tmp.append( t[j] )
            j = j + 1
        elif j > max :
            tmp.append( t[i] )
            i = i + 1
        elif t[i] < t[j] :
            tmp.append( t[i] )
            i = i + 1
        else :
            tmp.append( t[j] )
            j = j + 1
    for i in range( len(tmp) ):
        t[min+i] = tmp[i]

def tri_fusion_recursif( t, i, j ):
    if( i < j ):
        milieu = (i+j)//2
        tri_fusion_recursif( t, i, milieu )
        tri_fusion_recursif( t, milieu+1, j )
        fusion( t, i, milieu, j )

def tri_fusion( t ):
    tri_fusion_recursif( t, 0, len(t)-1 )
```

La complexité de ce programme est majoré par $c.n.\ln(n)$, où n est la taille du tableau et c un constante. Pour des tableaux contenant beaucoup d'éléments, cet algorithme est le le plus rapide parmi les algorithmes précédemment présentés.

11 Arbres

11.1 Arbres

Un arbre est une structure de données particulière que nous allons décrire sur un exemple particulier, avant de donner une définition formelle.

Un arbre est une structure de données contenant des sommets, deux à deux distincts. Dans la figure 17, les sommets sont représentés par des cercles contenant des entiers : les entiers 1, 2, 3, 5, 7, 10 et 11.

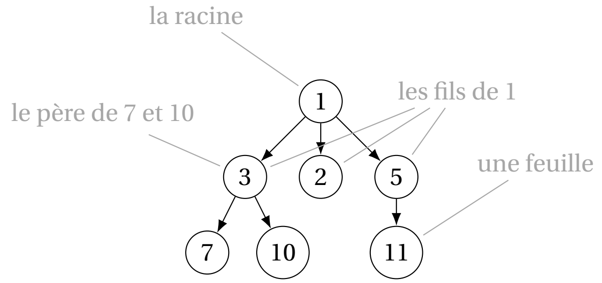


FIGURE 17 – Un exemple d'arbre

À l'exception d'un seul sommet, que l'on appelle *racine* , tous les sommets possèdent un unique *père* qui est un autre sommet de l'arbre.

Dans le dessin, la relation "père" est symbolisée par une flèche où, lorsqu'une flèche relie un sommet u à un sommet v, alors cela veut dire que le sommet u est le père de v. On dit aussi que v est le *fils* de u. Ainsi, toujours dans la figure 17, le sommet 1 est le père des sommets 2, 3 et 5. Le sommet 3, quand à lui, a pour fils les sommets 7 et 10.

Enfin, pour qu'une structure de données soit un arbre, il faut que, pour chaque sommet s, il existe toujours un chemin partant de la racine, terminant en s et empruntant les flèches de l'arbre.

Lorsque l'on dessine les arbres, la racine est le sommet qui n'est pointé par aucune flèche. Il est généralement dessiné tout en haut de la figure. Dans la figure, il s'agit du sommet 1.

Enfin, les sommets qui n'ont pas de fils sont appelés les *feuilles* de l'arbre. Dans la figure 17, les sommets 2, 7, 10 et 11 sont des feuilles.

Exercice 15

Listez le père et les fils de chaque sommet de l'arbre A de la figure 18. Donnez la racine et les feuilles de cet arbre.

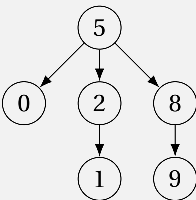


FIGURE 18 – L' arbre A

Passons maintenant à la définition formelle d'un arbre.

Un *arbre* est une structure de données A contenant des éléments, deux à deux distincts, appelés *sommets* munis des fonctions suivantes :

— la fonction fils(A, p) qui prends en paramètre un arbre A, un sommet p et renvoie une liste, éventuellement vide, de sommets de A; ces sommets sont appelés les fils de p;

— la fonction pere(A, f) qui prends en paramètre un arbre A , un sommet f et renvoie un sommet de A ou None; ce sommet est appelé le père de f;

— la fonction racine(A) qui prends en paramètre un arbre A et renvoie un sommet de A appelé *racine* de l'arbre.

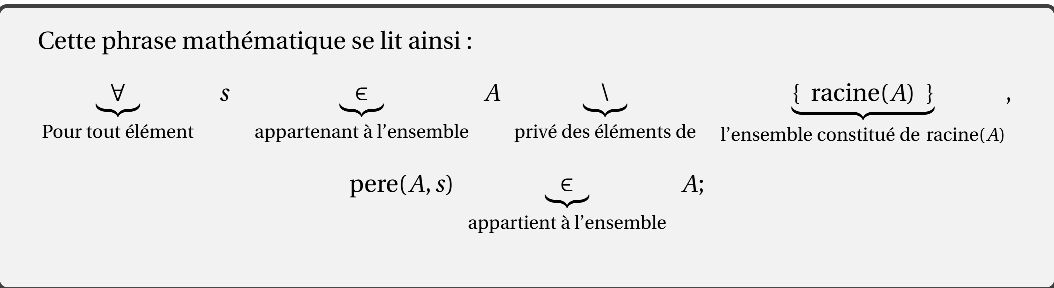
Ces opérations vérifient les propriétés suivantes :

Règle 1. Une racine n'a pas de père :

$$\text{pere}(A, \text{racine}(A)) = None;$$

Règle 2. Tout sommet possède un père à l'exception de la racine :

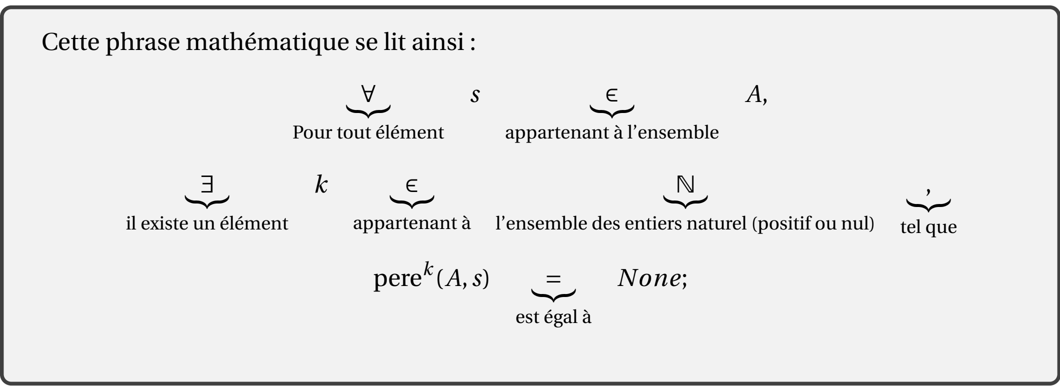
$$\forall s \in A \setminus \{\text{racine}(A)\}, \quad \text{pere}(A, s) \in A;$$



Règle 3. A partir de tout sommet, on peut accéder à la racine par la relation de paternité :

$$\forall s \in A, \exists k \in \mathbb{N}, \;\; \text{pere}^k(A, s) = None;$$

où pere^k se définit récursivement par : $\text{pere}^0(A, s) = s$ et $\text{pere}^{k+1}(A, s) = \text{pere}(A, \text{pere}^k(A, s))$ pour $k \geq 1$.



Règle 4. Le père d'un sommet f a pour fils f et réciproquement :

$$\forall p \in A, \; \forall f \in A, \;\; p = \text{pere}(A, f) \iff f \in \text{fils}(A, p).$$

Cette phrase mathématique se lit ainsi :

$$\underbrace{\forall}_{\text{Pour tout élément}} \quad p \quad \underbrace{\in}_{\text{appartenant à l'ensemble}} \quad A,$$

$$\underbrace{\forall}_{\text{pour tout élément}} \quad f \quad \underbrace{\in}_{\text{appartenant à l'ensemble}} \quad A,$$

$$p \underbrace{=}_{\text{est égal à}} \text{pere}(A, f) \quad \underbrace{\iff}_{\text{est équivalent à}} \quad f \quad \underbrace{\in}_{\text{appartient à l'ensemble}} \quad \text{fils}(A, p);$$

On peut récrire cette dernière phrase, de manière équivalente, ainsi :

Pour tout sommet p de A, pour tout sommet f de A, dire que p est le père de f est équivalent à dire que f est un fils de p.

On appelle arbre étiqueté un arbre A muni de l'opération etiquette(A, s) qui a un sommet s renvoie un élément appelé étiquette de s.

On dessine les étiquettes avec des rectangles reliés à leurs sommets respectifs par un trait. La figure 19 montre le dessin d'un arbre étiqueté.

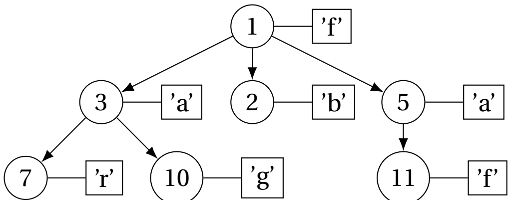


FIGURE 19 – Un exemple d'arbre étiqueté

Ainsi, dans la figure 19, le sommet 1 et le sommet 11 possèdent la même étiquette 'f' alors que le sommet 2 est étiqueté par 'b'.

Pour finir, voici la définition de l'arbre (que l'on notera A) de la figure 19, donné à l'aide des fonctions pere(), fils(), racine() et etiquette() :

$$\begin{cases} \text{racine}(A) = 1, \\ \text{pere}(A, \bullet) = \{1 : None, \quad 2:1 \quad 3:1 \quad 5:1 \quad 7:3 \quad 10:3 \quad 11:5\}, \\ \text{fils}(A, \bullet) = \{1 : [3,2,5], \quad 2:[] \quad 3:[7,10] \quad 5:[11] \quad 7:[] \quad 10:[] \quad 11:[]\}, \\ \text{etiquette}(A, \bullet) = \{1:\text{'f'}, \quad 2:\text{'b'}, \quad 3:\text{'a'}, \quad 5:\text{'a'}, \quad 7:\text{'r'}, \quad 10:\text{'g'}, \quad 11:\text{'f'}\}. \end{cases}$$

Exercice 16

Dessinez l'arbre correspondant à la définition mathématique suivante :

$$\begin{cases} \text{racine}(A) = 4, \\ \text{pere}(A, \bullet) = \{3:4, \quad 4:None \quad 5:3 \quad 8:3 \quad 9:5 \quad 10:5 \quad 11:8 \quad 12:11\}, \\ \text{fils}(A, \bullet) = \{3:[5,8], \quad 4:[3] \quad 5:[9,10] \quad 8:[11] \quad 9:[] \quad 10:[] \quad 11:[12] \quad 12:[]\}, \\ \text{etiquette}(A, \bullet) = \{3:\text{'a'}, \quad 4:\text{'b'}, \quad 5:\text{'a'}, \quad 8:\text{'a'}, \quad 9:\text{'f'}, \quad 10:\text{'g'}, \quad 11:\text{'f'}, \quad 12:\text{'g'}\}. \end{cases}$$

Exercice 17

Donnez la définition mathématique de l'arbre B dessiné à la figure 20.

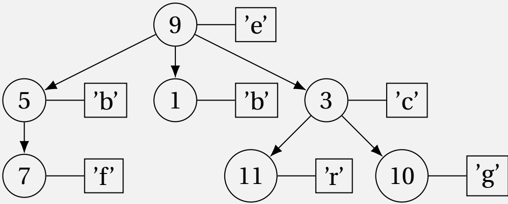


FIGURE 20 – L'arbre B

Exercice 18

1. Peut-on construire une structure de donnée, qui soit un arbre, et telle que la table des pères possède plusieurs fois la valeur None. Justifier votre réponse.

2. Peut-on construire une structure de données qui soit un arbre et telle qu'il existe deux chemins, partant de la racine, suivant les flèches et arrivant sur le même sommet ? Justifier votre réponse.
3. Peut-on avoir deux sommets qui ne soient la fin d'aucune flèche ? Justifier votre réponse.
4. Est-ce que la quatrième règle est équivalente à l'assertion suivante :

$$\forall s \in A, \;\; \forall f \in \text{fils}(A, s), \;\; \text{pere}(A, f) = s.$$

   Justifiez votre réponse.
5. Même question pour :

$$\forall s \in A, \;\; \forall f \in A, \;\; s = \text{pere}(A, f) \implies f \in \text{fils}(A, s).$$

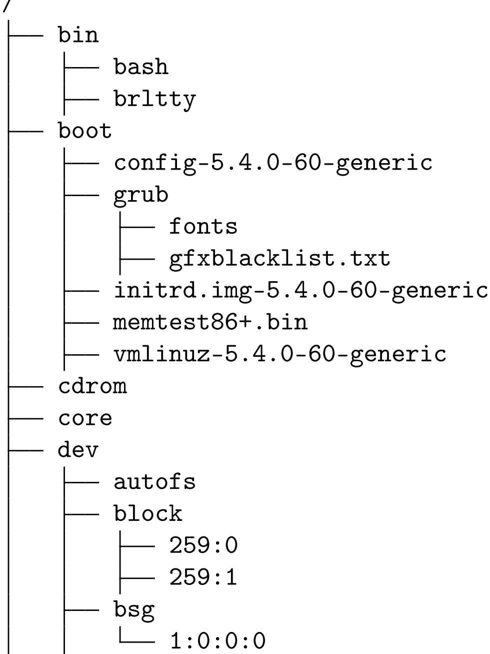
Exercice 19
On a vu que la structure de données d'un arbre est définie à l'aide des fonctions racine(), pere(), fils() et etiquette().
Y-a-t-il une redondance d'informations ? Dit autrement, peut on se passer de certaines fonctions, sans perdre d'information ?
Si oui, pourquoi et quelle jeu de fonctions faut-il garder, à minima, pour définir un arbre et pouvoir reconstruire les autres fonctions manquantes.

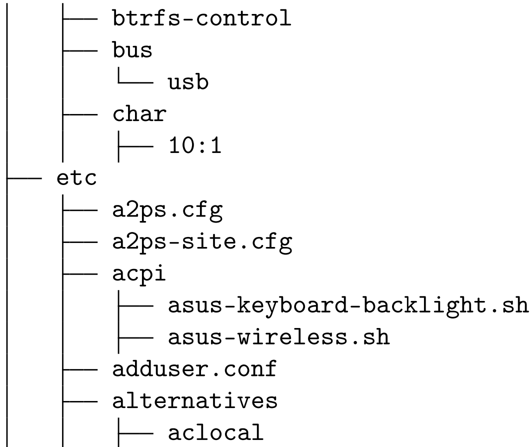
11.2 Exemples d'arbres en informatique

Les arbres sont utilisés très souvent en informatique.

Le système de fichiers est un exemple d'arbre. Les dossiers représentent des noeuds de l'arbre ou des feuilles quand ils sont vides. Les fichiers représentent des feuilles.

Voici, sous linux, un fragment de l'arborescence du système de fichier :





Exercice 20
Dans l'arborescence de fichiers ci-dessus, donnez quelques exemples de noeuds et de feuilles.

Exercice 21
Dessinez ensuite l'arborescence de votre dossier personnel (home). Sous Linux, vous pouvez vous aider de la commande 'tree' pour faire cela.

Les arbres sont aussi très utilisés dans les formats de fichiers. Par exemple, le format des codes sources des pages HTML utilise une structure d'arbre. Le code HTML suivant :

```
<html>
  <head>
    <meta name="robots" content="all" />
  </head>
  <body>
    <p>Bonjour</p>
    <p>Du texte</p>
  </body>
</html>
```

peut être représenté par l'arbre de la figure 21.

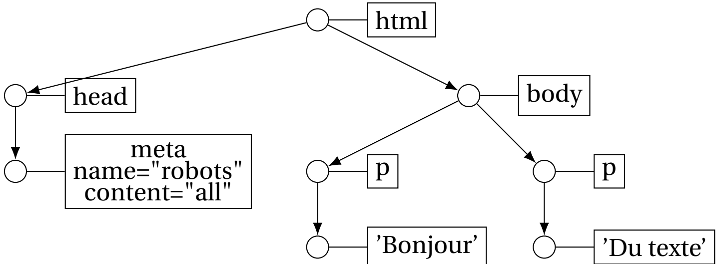


FIGURE 21 – L'arbre d'une page WEB en HTML.

Chaque balise HTML est alors converti en un noeud de l'arbre, muni d'une étiquette contenant les données de la balise..

Dans la figure 21 les numéros des sommets ne sont pas représentés. En effet, généralement, lorsque la page web est lue par l'ordinateur, puis chargée en mémoire, les sommets avec leurs étiquettes deviennent des objets alloués en mémoire. Les numéros des sommets sont alors implicites car il s'agit de l'adresse en mémoire où se trouve le sommet. En fait, on pourrait faire de même avec le code source de la page web et considérer, pour nous humain, que le numéro du sommet est le numéro de la ligne où commence la balise associée à ce même sommet. Dans les deux cas, cette numérotation n'apporte pas beaucoup d'information et est donc toujours omise.

Exercice 22
Dessinez (partiellement) l'arbre d'une page html. Par exemple, vous pouvez choisir la page web suivante, qui n'est pas trop compliquée : `https://www.labri.fr/perso/boussica/bibliotheque_graphique_fr.html`
Pour récupérer le code source de la page, vous pouvez, sous firefox, faire un clique droit et cliquer sur "View Page Source". Vous pouvez aussi télécharger la page est l'ouvrir avec un éditeur de texte.

11.3 Une API pour les arbres

Voici une API possible pour créer des arbres :

```
def creer_arbre(racine=None, etiquette=None):
  # Créer un arbre contenant la racine `racine`.
  # si racine vaut None, on considère l'arbre vide.
def creer_sommet(A=None, numero=None):
  # Créer et retourne un sommet ayant pour numéro 'numero'
def inserer_racine(A, racine, etiquette=None):
  # ajoute la racine `racine` dans l'arbre et lui associe
  # l'étiquette `etiquette`.
def inserer_fils(A, pere, fils, etiquette=None):
  # Dans l'arbre `A`, ajoute le sommet `fils`
  # dans la liste des fils du sommet `pere`.
  # Le sommet `fils` aura pour étiquette `etiquette`.
```

Voici une API possible pour manipuler les arbres :

```
def fils(A, p)
    # renvoie la liste des fils du sommet p dans l'arbre A
def pere(A, f)
    # renvoie le père du sommet f dans l'arbre A, ou None
    # si f est la racine
def racine(A)
    # renvoie la racine de l'arbre A
def etiquette(A, f):
    # renvoie l'étiquette de f dans A
```

Et voici une implémentation possible de ces deux API :

```
def creer_arbre_vide():
  return {
    'racine' : None,
    'etiquettes' : {},
    'fils' : {},
    'pere' : {}
```

```
    }

def creer_sommet(A=None, numero=None):
    if numero is None:
        raise ValueError("Numero manquant")
    return numero

def inserer_fils(A, pere, fils, etiquette=None):
    if not pere is None:
        A['fils'][pere].append(fils)
    A['pere'][fils] = pere
    A['fils'][fils] = []
    A['etiquettes'][fils] = etiquette

def inserer_racine(A, racine, etiquette=None):
    A['racine'] = racine
    if not racine is None:
        inserer_fils(A, None, racine, etiquette)

def creer_arbre(racine=None, etiquette=None):
    res = creer_arbre_vide()
    inserer_racine(res, racine, etiquette)
    return res

def racine(A):
    return A['racine']

def fils(A, s):
    return A['fils'][s]

def pere(A, s):
    return A['pere'][s]

def etiquette(A,s):
    return A['etiquettes'][s]
```

Exercice 23
Testez la bibliothèque précédente en implémentant l'arbre B de la figure 20.

Exercice 24
L'implémentation proposée ci-dessus permet-elle de construire des structures de données mal formées, c'est à dire qui ne vérifient pas les 4 propriétés associées aux arbres ?
Si oui, modifiez le code pour interdire (en levant une erreur) toute modification de l'arbre qui produirait un arbre mal formé.

La plupart du temps, dans les bibliothèques, les arbres sont implémentés différemment. Les sommets des arbres sont des structures de données. La table des pères et des fils sont codées directement dans les sommets de l'arbre à l'aide des attributs 'pere' et 'fils' qui contiennent des références respectivement au sommet père et aux sommets fils. Le numéro d'un sommet est alors implicite car il s'agit

alors de l'adresse en mémoire (référence en python) où se trouve la structure de données représentant le sommet. La figure 22 donne la représentation mémoire de ce type d'arbre.

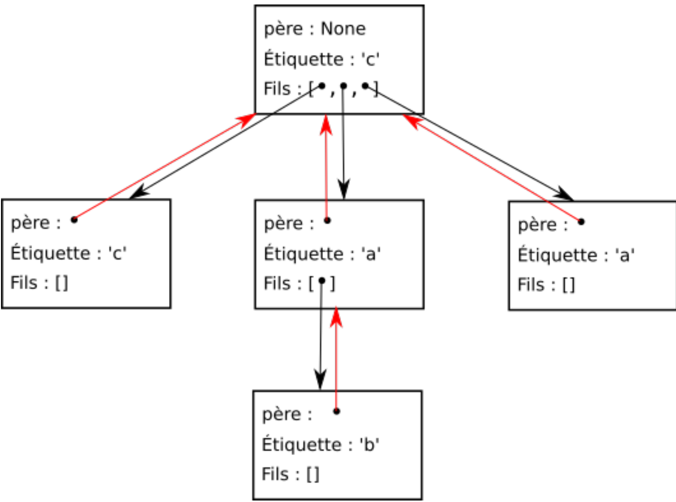


FIGURE 22 – Une structure de données classique d'arbre

Cette implémentation des arbres est préférée à la première car elle se prête mieux à l'ajout et à la suppression de noeuds. En effet, il n'est pas nécessaire de gérer les numéros des sommets. De plus, les noeuds peuvent être ajoutés n'importe où en mémoire sans avoir à modifier profondément les tables existantes.

Exercice 25
Proposer une implémentation de l'API précédente, où

— les relations père et fils sont implémentées directement dans les sommets de l'arbre

— la numérotation des sommets est omise car remplacée par l'adresse mémoire où se situe l'objet.

Vous testerez votre nouvelle bibliothèque d'arbre en implémentant l'arbre de la figure 22. Utilisez PythonTutor pour inspecter la mémoire, afficher l'arbre et vérifier que votre arbre est bien formé.

11.4 Arbres binaires

Tout comme à la section précédente, nous allons présenter les arbres binaires sur un exemple avant de donner une définition formelle.

Un arbre binaire est "un arbre" un peu particulier. Pour commencer, c'est un arbre où chaque sommet possède 0,1 ou 2 fils, comme vous pouvez l'observer à la figure 23.

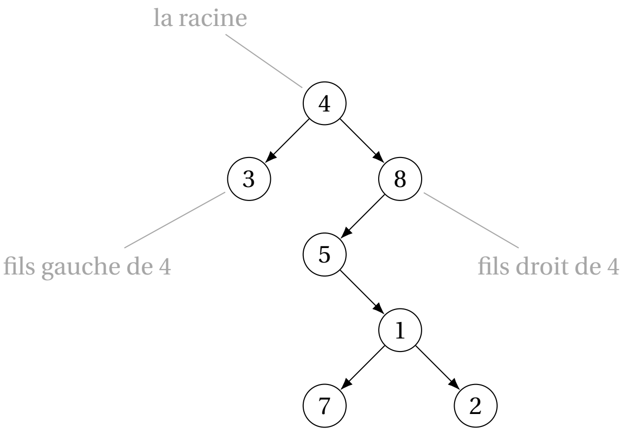


FIGURE 23 – Un arbre binaire

Cependant, contrairement aux arbres, les fils sont nommés. Ainsi, un fils peut être, soit un fils gauche et on le dessine à gauche de son père, soit un fils droit, et on le dessine à droite de son père. Ainsi, dans la figure 23, le sommet 4 possède un fils gauche qui est le sommet 3 et un fils droit qui est le sommet 8. De même, toujours à la figure 23, le sommet 5 ne possède pas de fils gauche. Par contre, il possède un fils droit qui est le sommet 1.

Il est important de remarquer que les arbres binaires ne sont pas des arbres particuliers. En effet nommer explicitement les fils change la structure de l'objet. Par exemple, si vous regardez les deux arbres binaires de la figure 24, alors les deux arbres binaires A_1 et A_2 sont différents car dans A_1, le sommet 2 est le fils gauche du sommet 1 alors que dans l'arbre A_2, le sommet 2 est sont fils droit. Cependant, si l'on interprète A_1 et A_2 comme des dessins d'arbres, alors A_1 et A_2 représentent le même arbre (où les fils se sont pas ordonnés).

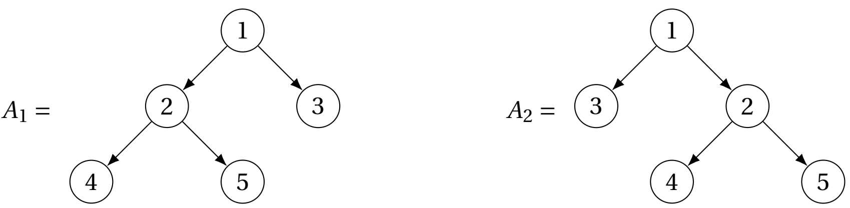


FIGURE 24 – Deux arbres binaires A_1 et A_2 différents

La figure 25 montre encore deux arbres binaires différents qui ont le même arbre (que les fils sont ordonnées ou pas ordonnées).

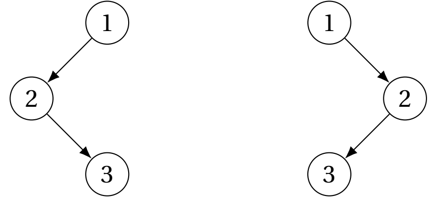


FIGURE 25 – Deux arbres binaires différents

Nous pouvons maintenant donner une définition formelle des arbres binaires.

Un arbre binaire est une structure de données A contenant des éléments, deux à deux distincts, appelés *sommets* et munie des fonctions suivantes :

— la fonction fils_gauche(A, p) qui prends en paramètre un arbre A, un sommet p et renvoie un sommet de A appelé fils gauche de p ou $None$ si p n'a pas de fils gauche.

— la fonction fils_droit(A, p) qui prends en paramètre un arbre A, un sommet p et renvoie un sommet de A appelé fils droit de p ou $None$ si p n'a pas de fils droit.

— la fonction pere(A, f) qui prends en paramètre un arbre A , un sommet f et renvoie un sommet de A ou None ; ce sommet est appelé le père de f ;

— la fonction racine(A) qui prends en paramètre un arbre A et renvoie un sommet de A appelé racine de l'arbre.

Ces opérations vérifient les propriétés suivantes :

— Une racine n'a pas de père :

$$\text{pere}(A, \text{racine}(A)) = None;$$

— Tout sommet possède un père à l'exception de la racine :

$$\forall s \in A \setminus \{\text{racine}(A)\}, \ \ \text{pere}(A, s) \in A;$$

— A partir de tout sommet, on peut accéder à la racine par la relation de paternité :

$$\forall s \in A, \exists k \in \mathbb{N}, \ \ \text{racine}(\text{pere}^k(A, s));$$

où pere^k se définit récursivement par : $\text{pere}^0(A, s) = s$ et $\text{pere}^{k+1}(A, s) = \text{pere}(A, \text{pere}^k(A, s))$ pour $k \geq 1$.

— Les fils gauches d'un sommet s ont pour père s :

$$\forall s \in A, \quad \text{si} \quad \text{fils_gauche}(A, s) \neq None \quad \text{alors} \quad \text{pere}(A, \text{fils_gauche}(A, s)) = s;$$

— Les fils droits d'un sommet s ont pour père s :

$$\forall s \in A, \quad \text{si} \quad \text{fils_droit}(A, s) \neq None \quad \text{alors} \quad \text{pere}(A, \text{fils_droit}(A, s)) = s.$$

On appelle arbre binaire étiqueté un arbre binaire A muni de l'opération etiquette(A, s) qui a un sommet s renvoie un élément appelé étiquette de s.

Sans grande nouveauté, on dessine les étiquettes des arbres binaires comme on les dessine pour les arbres. La figure 26 montre un exemple d'arbre binaire étiqueté.

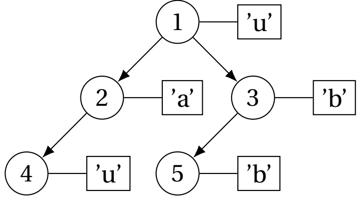


FIGURE 26 – Un arbre binaire étiqueté.

Enfin, pour finir, voici la définition de l'arbre étiqueté (que l'on notera B) de la figure 26 :

$$\begin{cases} \text{racine}(B) = 1; \\ \text{pere}(B, \bullet) = \{1 : None, \ \ 2 : 1, \ \ 3 : 1, \ \ 4 : 2, \ \ 5 : 3\}; \\ \text{fils_gauche}(B, \bullet) = \{1 : 2, \ \ 2 : 4, \ \ 3 : 5, \ \ 4 : None, \ \ 5 : None\}; \\ \text{fils_droit}(B, \bullet) = \{1 : 3, \ \ 2 : None, \ \ 3 : None, \ \ 4 : None, \ \ 5 : None\}; \\ \text{etiquette}(B, \bullet) = \{1 : \text{'u'}, \ \ 2 : \text{'a'}, \ \ 3 : \text{'b'}, \ \ 4 : \text{'u'}, \ \ 5 : \text{'b'}\}; \end{cases}$$

Exercice 26
Voici la définition d'un arbre binaire :

$$\begin{cases} \text{racine}(A) = 1; \\ \text{pere}(A, \bullet) = \{1 : None, \quad 4:1, \quad 5:4, \quad 8:5, \quad 9:5\}; \\ \text{fils_gauche}(A, \bullet) = \{1:4, \quad 4: None, \quad 5:8, \quad 8: None, \quad 9: None\}; \\ \text{fils_droit}(A, \bullet) = \{1: None, \quad 4:5, \quad 5:9, \quad 8: None, \quad 9: None\}; \\ \text{etiquette}(A, \bullet) = \{1: \text{'u'}, \quad 4: \text{'a'}, \quad 5: \text{'b'}, \quad 8: \text{'g'}, \quad 9: \text{'a'}\}. \end{cases}$$

Dessiner l'arbre binaire associé.

Exercice 27
Donnez la définition mathématique de l'arbre binaire de la figure 27.

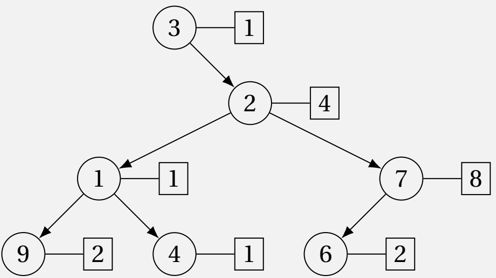


FIGURE 27 – L'arbre binaire C.

Exercice 28
Un arbre binaire saturé A est un arbre dont tous les sommets possèdent 0 ou 2 fils et telle que les feuilles de l'arbre sont toujours à la même distances de la racine, c'est à dire qu'il existe un entier h, appelé hauteur de l'arbre, telle que, pour toute feuille f, $\text{pere}^h(A, f) = \text{racine}(A)$.

1. Dessinez des exemples d'arbre binaires saturés ayant, pour hauteurs respectifs, $h = 0$, $h = 1$, $h = 2$ et $h = 3$.
2. En connaissant la valeur de h, pouvez-vous donner une formule, qui dépend de h, qui permet de déterminer le nombre de sommets que possède l'arbre A?

Généralement, la numérotation des sommets des arbres ne nous intéresse pas beaucoup. Elle est utilisée uniquement pour pouvoir désigner un sommet particulier. Ainsi, il n'est pas rare de représenter les arbres binaires sans le numéro de leurs sommets. Par exemple, la figure 28, montre un arbre binaire où les sommets ne sont pas numérotés.

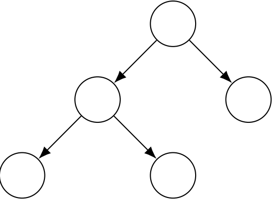


FIGURE 28 – Un exemple d'arbre binaire où les sommets ne sont pas numérotés.

Ainsi, on dit que deux arbres sont équivalents, ou homéomorphe, si ils sont identiques, à renumérotation des sommets prêt. Quand on s'intéresse aux arbres à renumérotation des sommets prêts, alors on ne numérote pas les sommets de l'arbre. La figure 29 montre deux exemples d'arbres binaires équivalents à renumérotation des sommets prêts, ainsi que sa représentation sans numéro de sommet.

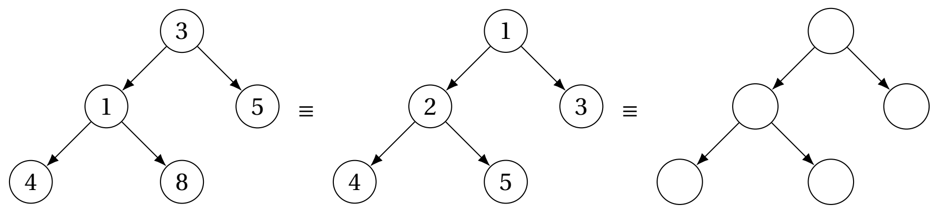


FIGURE 29 – Ces arbres binaires sont équivalents à renumérotation des sommets prêts.

Exercice 29

1. Combien y-a-t-il d'arbres binaires à renumérotation des sommets prêt, contenant respectivement 0, 1, 2, 3 et 4 sommets? Dessinez ces arbres binaires.
2. (très TRÈS **TRÈS DIFFICILE** – à ne faire que par passion) Donnez une formule, en fonction du nombre de sommets, qui donne le nombre d'arbres binaires obtenus à renumérotation des sommets prêts (indication pour obtenir la solution : consultez le site `https://oeis.org`).
3. Que dire d'un algorithme, qui, pour trouver une solution, passerait en revue tous les arbres binaires à n sommets?

11.5 Une API pour les arbres binaires

Voici une API possible pour créer les arbres binaires :

```
def creer_arbre_binaire(racine=None, etiquette=None):
    # Créer un arbre binaire contenant la racine `racine`.
    # si racine vaut None, on considère l'arbre vide.
def inserer_racine(A, racine, etiquette=None):
  # ajoute la racine `racine` dans l'arbre et muni la
  # racine de l'etiquette `etiquette`.
def inserer_fils_gauche(A, pere, fils, etiquette=None):
  # Dans l'arbre `A`, ajoute le sommet `fils`
  # comme fils gauche de `pere`.
  # Le sommet `fils` aura pour etiquette `etiquette`.
def inserer_fils_droit(A, pere, fils_droit, etiquette=None):
```

```
# Dans l'arbre `A`, ajoute le sommet `fils`
# comme fils droit de `pere`.
# Le sommet `fils` aura pour etiquette `etiquette`.
```

Voici une API possibile pour les manipuler :

```
def fils_droit(A, p)
    # renvoie le fils droit du sommet p dans l'arbre A
def fils_gauche(A, p)
    # renvoie le fils gauche du sommet p dans l'arbre A
def pere(A, f)
    # renvoie le père du sommet f dans l'arbre A, ou None
    # si f est la racine
def racine(A)
    # renvoie la racine de l'arbre A
def etiquette(A, f):
    # renvoie l'etiquette de f dans A
```

Et voici une implémentation possible de ces deux API :

```
def creer_arbre_vide():
  return {
    'racine' : None,
    'etiquettes' : {},
    'fils_droit' : {},
    'fils_gauche' : {},
    'pere' : {}
  }

def inserer_ab(A, pere, fils, etiquette=None, type_fils=None):
  if fils is None:
    return
  if not pere is None:
    A[type_fils][pere] = fils
  A['pere'][fils] = pere
  A['fils_gauche'][fils] = None
  A['fils_droit'][fils] = None
  A['etiquettes'][fils] = etiquette

def inserer_fils_gauche(A, pere, fils, etiquette=None):
  inserer_ab(A, pere, fils, etiquette=etiquette, type_fils='fils_gauche')

def inserer_fils_droit(A, pere, fils, etiquette=None):
  inserer_ab(A, pere, fils, etiquette=etiquette, type_fils='fils_droit')

def inserer_racine(A, racine, etiquette=None):
  A['racine'] = racine
  inserer_ab(A, None, racine, etiquette, type_fils=None)

def creer_arbre(racine=None, etiquette=None):
  res = creer_arbre_vide()
  inserer_racine(res, racine, etiquette)
  return res

def racine(A):
```

```
    return A['racine']

def fils_gauche(A, s):
    return A['fils_gauche'][s]

def fils_droit(A, s):
    return A['fils_droit'][s]

def pere(A, s):
    return A['pere'][s]

def etiquette(A,s):
    return A['etiquettes'][s]
```

Exercice 30
Testez la bibliothèque précédente en implémentant l'arbre binaire étiqueté de la figure 26.

Exercice 31
Reprenez l'exercice 25 en l'adaptant pour les arbres binaires. Proposez une figure (comme celle de la figure 22) pour illustrer votre propos et votre implémentation.

11.6 Quelques définitions pour les arbres et les arbres binaires

On appelle *taille d'un arbre* le nombre de sommets de l'arbre. Par exemple, la taille de l'arbre de la figure 30 est 7.

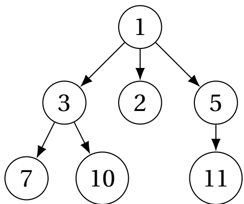


FIGURE 30 – Un exemple d'arbre de taille 7 et de hauteur 2

Dans un arbre A, la *hauteur d'un sommet* s est l'entier h tel que $pere^h(A,s) = racine(A)$. On dit aussi que la *hauteur d'un sommet* est la distance (le nombre d'arête) entre sa racine et lui. Par exemple, dans la figure 30, la hauteur du sommet 3 est 1, la hauteur du sommet 10 est 2 et la hauteur de la racine 1 est 0.

On appelle hauteur de l'arbre, la hauteur maximale de ses sommets. Ainsi, la hauteur de l'arbre de la figure 30 est 2.

On dit qu'un sommet s est un *ancêtre* de t ou bien que t est un *descendant* de s si il existe un entier $k \geq 0$ tel que : $pere^k(A,t) = s$.

Par exemple, toujours dans la figure 30, le sommet 7 est le descendant de 1 et le sommet 1 est l'ancêtre de 7. De même, le sommet 3 est le descendant et l'ancêtre de lui même. Attention, selon les définitions (et donc selon les livres et articles), un sommet ne peut pas être le descendant (resp. l'ancêtre) de lui même. Dans ce cours, le choix qui a été fait, est de considérer q'un sommet est toujours le descendant (resp. ancêtre) de lui-même.

On appelle *sous-arbre* de A de sommet s l'arbre A' constitué uniquement des descendants de s dans A, en restreignant la fonction fils aux descendant s de A et en définissant s comme étant la nouvelle racine de ce sous-arbre.

Par exemple, dans la figure 31, l'arbre A', qui est à gauche, est un sous-arbre de sommet 3 de l'arbre A, qui est à droite.

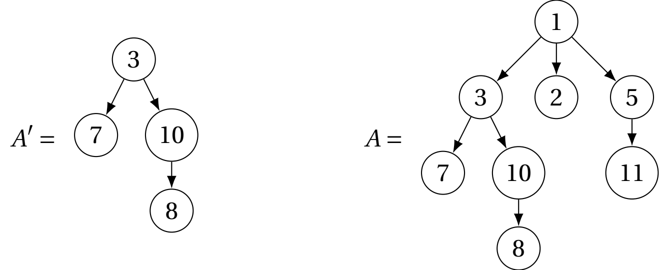


FIGURE 31 – L'arbre A' est un sous-arbre de sommet 3 de l'arbre A.

Exercice 32
Dans cet exercice on considère l'arbre de la figure 32.

1. Donnez la hauteur de cette arbre ainsi que la hauteur de chacun de ses sommets.
2. Pour chacun de ses sommets, donnez leurs ancêtres ainsi que leurs descendants.
3. Pour chacun des ses sommets s donnez les sous-arbres de racine s.

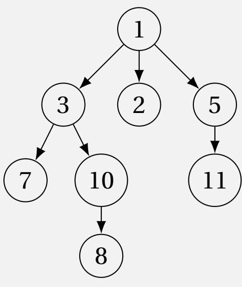


FIGURE 32 – Un arbre.

Exercice 32.1 Écrivez un programme, qui utilise uniquement l'API des arbres et qui renvoie la taille d'un arbre.

Exercice 32.2 Écrivez un programme, qui utilise uniquement l'API des arbres et qui renvoie la hauteur d'un sommet d'un arbre.

Exercice 32.3 Écrivez un programme, qui utilise uniquement l'API des arbres et qui renvoie la hauteur d'un arbre.

11.7 Parcours d'un arbre

Il existe plusieurs façons de parcourir les sommets d'un arbre. Dans ce cours, nous allons voir

— 3 parcours pour les arbres et les arbres binaires : le parcours en largeur, le parcours préfixe et le parcours postfixe ;

— 1 parcours pour les arbres binaires uniquement : le parcours infixe.

Un parcours est une façon de passer en revue une et une seule fois tous les sommets de l'arbre. Un parcours est donc représenté par une liste des sommets du graphe.

Commençons par le parcours en largeur.

Le *parcours en largeur* est une liste de sommets ordonnée selon la hauteur des sommets.

Par exemple, un parcours en largeur possible de l'arbre de la figure 33 est $[1, 3, 2, 5, 7, 10, 11, 8]$.

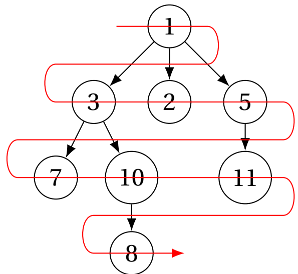
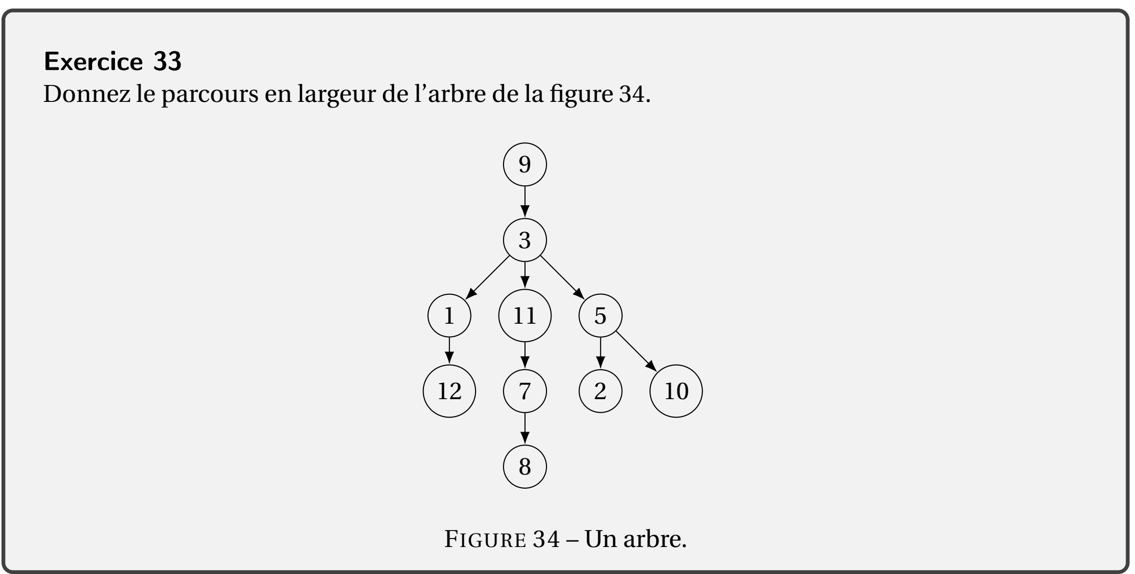


FIGURE 33 – Un parcours en largeur possible d'un arbre.



Exercice 33

Donnez le parcours en largeur de l'arbre de la figure 34.

FIGURE 34 – Un arbre.

Dans les arbres binaires il existe trois parcours supplémentaires : le parcours préfixe, infixe et postfixe.

Le parcours préfixe d'un arbre se définit récursivement par

$$\begin{aligned} \text{prefixe}(A) &= \text{prefixe}(A, \text{racine}(s)) \\ \text{prefixe}(A, s) &= [s] + \text{prefixe}(\text{fils_gauche}(A, s)) + \text{prefixe}(\text{fils_droit}(A, s)) \\ \text{prefixe}(A, None) &= []. \end{aligned}$$

Le parcours infixe d'un arbre se définit récursivement par

$$\begin{aligned} \text{infixe}(A) &= \text{infixe}(A, \text{racine}(s)) \\ \text{infixe}(A, s) &= \text{infixe}(\text{fils_gauche}(A, s)) + [s] + \text{infixe}(\text{fils_droit}(A, s)) \\ \text{infixe}(A, None) &= []. \end{aligned}$$

Le parcours postfixe d'un arbre se définit récursivement par

$$\begin{aligned}
\text{postfixe}(A) &= \text{postfixe}(A, \text{racine}(s)) \\
\text{postfixe}(A, s) &= \text{postfixe}(\text{fils_gauche}(A, s)) + \text{postfixe}(\text{fils_droit}(A, s)) + [s] \\
\text{postfixe}(A, None) &= [\,].
\end{aligned}$$

Comme l'illustre la figure 35, il est possible de représenter graphiquement ces trois parcours en traçant un chemin qui fait le tour de l'arbre en partant de la racine et en tournant dans le sens inverse des aiguilles d'une montre. Ensuite, on dessine

— pour le parcours préfixe, un trait juste à gauche de chaque sommet (cf. le dessin de gauche de la figure 35) ;

— pour le parcours infixe, un trait juste en dessous de chaque sommet (cf. le dessin du milieu de la figure 35) ;

— pour le parcours postfixe, un trait juste à droite de chaque sommet (cf. le dessin de droite de la figure 35).

Enfin, les parcours préfixe, resp. infixe et resp. postfixe sont obtenus en faisant le tour de l'arbre et en notant les sommets associés aux traits croisés durant le tour.

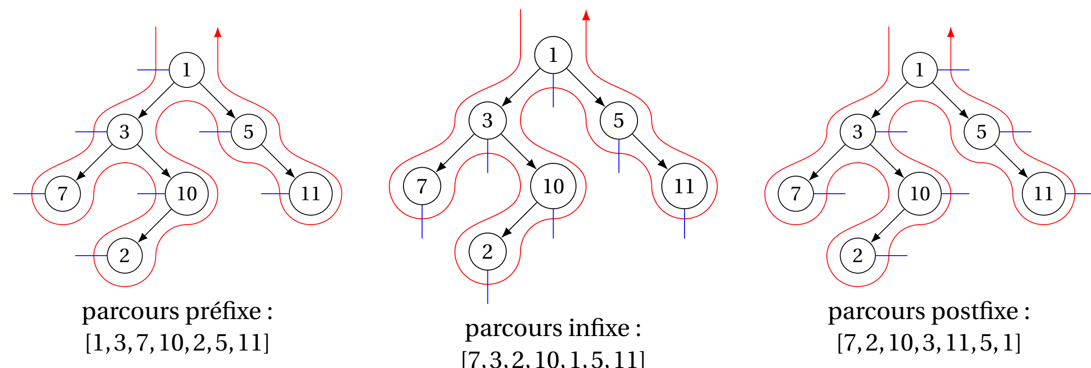
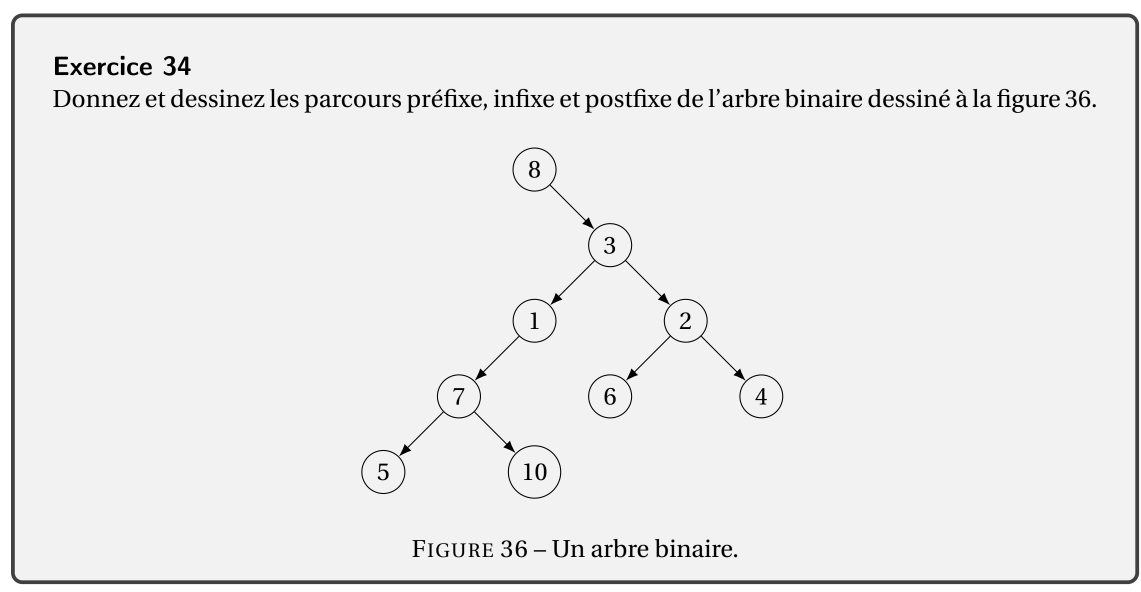


FIGURE 35 – Les parcours préfixe, infixe et postfixe d'un arbre binaire.



Exercice 34
Donnez et dessinez les parcours préfixe, infixe et postfixe de l'arbre binaire dessiné à la figure 36.

FIGURE 36 – Un arbre binaire.

Les parcours préfixe et postfixe peuvent aussi être définis pour les arbres si les fils sont ordonnés.

Le parcours préfixe d'un arbre se définit récursivement par :

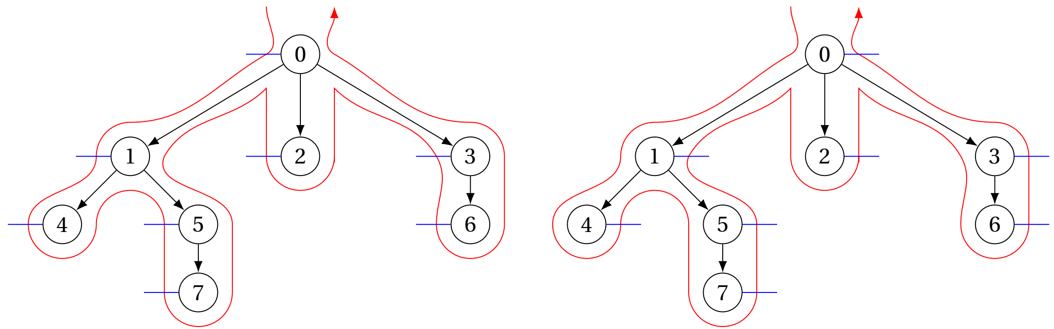
$$\begin{aligned}
\text{prefixe}(A) &= \text{prefixe}(A, \text{racine}(s)) \\
\text{prefixe}(A, s) &= [s] + \sum_{f \in \text{fils}(s)} \text{prefixe}(A, f) \\
\text{prefixe}(A, None) &= [].
\end{aligned}$$

où $\sum$ est l'opérateur de sommation que l'on définira sur les listes par $\sum_{f \in [2,5,3,8]} \text{fct}(f) = \text{fct}(2) + \text{fct}(5) + \text{fct}(3) + \text{fct}(8)$.

De même, le parcours postfixe d'un arbre se définit récursivement par :

$$\begin{aligned}
\text{postfixe}(A) &= \text{postfixe}(A, \text{racine}(s)) \\
\text{postfixe}(A, s) &= \left(\sum_{f \in \text{fils}(s)} \text{postfixe}(A, f) \right) + [s] \\
\text{postfixe}(A, None) &= [].
\end{aligned}$$

La figure 37 montre les parcours préfixe et infixe d'un arbre.



Parcours préfixe : [0, 1, 4, 5, 7, 2, 3, 6] Parcours postfixe : [4, 7, 5, 1, 2, 6, 3, 0]

FIGURE 37 – Les parcours préfixe et infixe d'un arbre.

Exercice 35

1. Donnez les parcours préfixe et postfixe de l'arbre de la figure 34.
2. Peut-on obtenir un parcours infixe pour cette arbre? Justifiez votre réponse.

Ces parcours jouent un rôle important en informatique. En effet, l'ordre d'exécution des appels de fonctions d'un programme est donné par le parcours préfixe de l'arbre d'exécution (cet ordre est le même que l'ordre des temps de début d'exécution des fonctions d'un programme). De même, l'ordre des temps de fins d'exécution des fonctions d'un programme est donné par le parcours postfixe de ce même arbre. Ces deux parcours jouent donc un rôle important lorsque l'on veut démontrer le fonctionnement d'un algorithme récursif où étudier la complexité d'un algorithme récursif.

Par exemple, étudions le programme f qui calcule récursivement la suite de Fibonacci de la manière suivante :

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

L'arbre des appels récursifs de fib(4) est dessiné à la figure 38. Dans cet arbre, lorsqu'une flèche relie un sommet s à un sommet t, cela signifie que l'exécution de la fonction présent dans l'étiquette de s a

fait appel à la fonction présent dans l'étiquette de t. De plus, les fils d'une même fratrie sont ordonnés (de gauche à droite dans le dessin) selon leurs ordres d'exécution dans le temps.

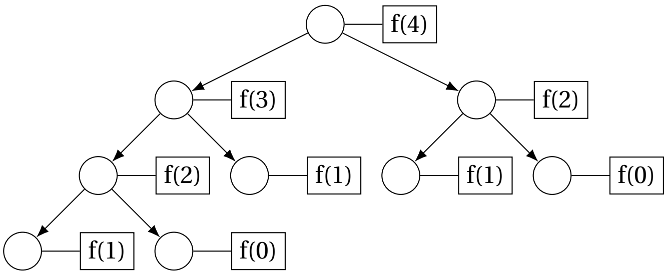


FIGURE 38 – L'arbre des appels récursifs de f(4).

Grâce à cet arbre, il est facile maintenant d'évaluer la complexité du programme qui cacule le nombre de Fibonacci. Il s'agit du nombre de sommets de l'arbre multiplié par un constante C où C est le nombre d'opérations, hors appels de fonctions, réalisées par f. Ainsi, comme l'arbre de f(4) contient 9 sommets, le programme f(4) réalise $9 \times C$ opérations, où C est une constante que l'on peut évaluer à environ une dizaine d'opérations.

Pour finir, nous verrons, à la section sur les arbres binaires de recherche (cf. section 11.8), que l'ordre infixe peut être utilisé pour trier un ensemble d'éléments par ordre croissant.

Exercice 36
Proposez trois programmes qui prends en paramètre un arbre binaire et renvoie une liste de sommets ordonnés dans l'orde des parcours préfixe, infixe et postfixe.

Exercice 37
Proposez deux programmes qui prends en paramètre un arbre et renvoie une liste de sommets ordonnés dans l'orde des parcours préfixe et postfixe.

Exercice 38
Proposez un programme qui prends en paramètre un arbre et renvoie une liste de sommets ordonnés dans l'orde du parcours en largeur.

11.8 Arbres binaires de recherche

Un arbre binaire de recherche est un arbre étiqueté par des réels où, pour tout sommet s de l'arbre, on a les propriétés suivantes :

— les étiquettes des sommets du sous-arbre gauche de s sont toutes plus petites ou égales que l'étiquette de s;

— les étiquettes des sommets du sous-arbre droit de s sont toutes strictement plus grandes que l'étiquette de s.

Par exemple, l'arbre de la figure 39 est un arbre binaire de recherche.

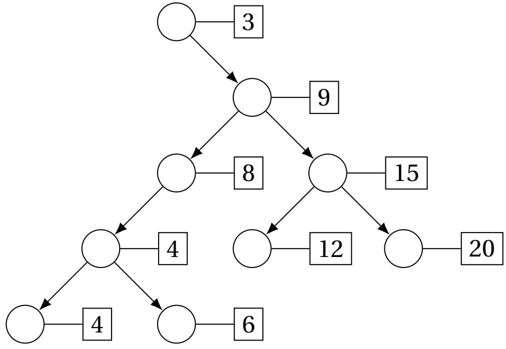


FIGURE 39 – Un arbre binaire de recherche.

A contratio, l'arbre de la figure 40 n'est pas un arbre binaire de recherche. En effet, le sommet s qui est dans le sous-arbre gauche de t possède une étiquette plus grande que celle de t.

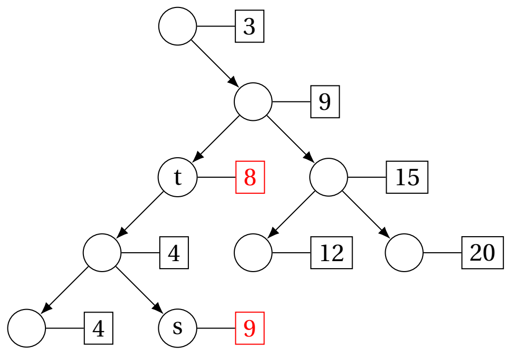


FIGURE 40 – Cet arbre n'est pas un arbre binaire de recherche.

Exercice 39

1. Donnez des exemples d'arbres binaires de recherche ;
2. Pour chacun des ces arbres, ainsi que pour l'arbre de la figure 39, ordonnez les étiquettes dans l'ordre du parcours infixe de leurs sommets. Que remarquez-vous ? Pourquoi ?
3. Donnez des exemples d'arbres binaires qui ne sont pas des arbres binaires de recherches.
4. Que se passe-t-il, cette fois-ci, lorsque vous ordonnez les étiquettes dans l'ordre du parcours infixe de leurs sommets ?

Les arbres binaires de recherche permettent de faire une recherche dichotomique dans un ensemble d'éléments avec répétition que l'on appelle généralement *multi-ensemble* .

L'idée est d'utiliser les étiquettes d'un arbre binaire de recherche pour coder un multi-ensemble.

Par exemple, l'arbre de la figure 39 encode le multi-ensemble $\{\{3,4,4,6,8,9,12,15,20\}\}$.

Il faut noter que plusieurs arbres binaires de recherche peuvent partager le même multi-ensemble. Par exemple, l'arbre de la figure 41 possède le même multi-ensemble que celui de la figure 39.

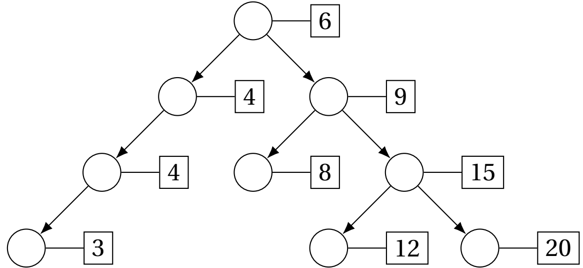


FIGURE 41 – Les étiquettes de cet arbre forme le même multi-ensemble que celui de la figure 39.

Pour chercher un élément dans un multi-ensemble codé par un arbre binaire de recherche, ils suffit de parcourir les sommets de l'arbre, en partant de la racine, et à chaque sommet, on se déplace (si c'est possible) sur le sommet de gauche si l'élément recherché est strictement plus petit que l'étiquette du sommet courant et sur le fils droit si l'élément est strictement plus grand. Le parcours s'arrête quand on trouve l'élément cherché dans l'étiquette du sommet courant ou qu'il n'est plus possible de se déplacer dans l'arbre. A la fin du parcours, le multi-ensemble contient l'élément cherché e si et seulement si le parcours se termine sur un sommet d'étiquette e.

Par exemple, l'arbre de la figure 42 code le multi-ensemble $\{\{2,2,8,9,10,12,13,14,14,16\}\}$ et la recherche dichotomique de 10 dans ce multi-ensemble est symbolisée par un chemin en rouge et gras dans l'arbre binaire de recherche. Comme le parcours associé se termine sur un sommet étiqueté 10, on en déduit que le multi-ensemble contient l'élément 10. Maintenant, si on cherche l'élément 11, on obtient le parcours symbolisé par un chemin en rouge et gras dans l'arbre de la figure 43. Comme le parcours se termine sur une feuille qui n'est pas étiquetée par 11, mais par 12, alors le multi-ensemble ne contient pas l'élément 11. Enfin, si on cherche l'élément 9, on obtient le parcours symbolisé par un chemin en rouge et gras dans l'arbre de la figure 43. Comme le parcours se termine sur un sommet qui n'est pas étiqueté par 9, mais par 10, alors le multi-ensemble ne contient pas l'élément 9.

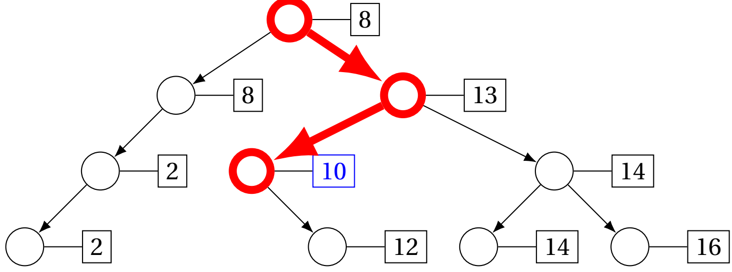


FIGURE 42 – La recherche fructueuse de l'élément 10 dans un arbre binaire de recherche.

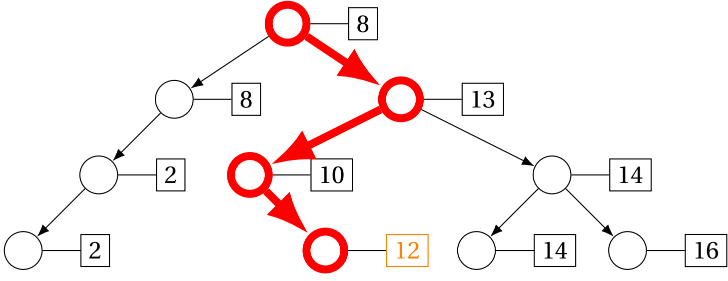


FIGURE 43 – La recherche infructueuse de l'élément 11 dans un arbre binaire de recherche.

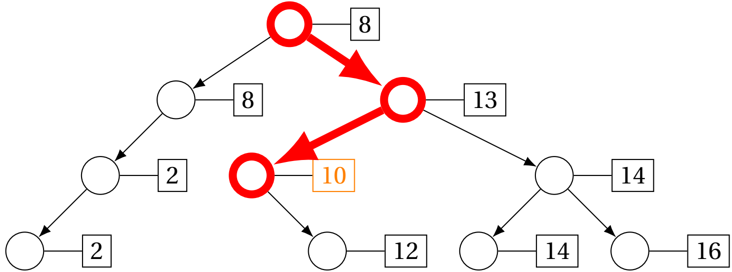


FIGURE 44 – La recherche infructueuse de l'élément 9 dans un arbre binaire de recherche.

Exercice 40
Appliquez l'algorithme de recherche dichotomique pour trouver l'élément 4 dans l'arbre de la figure 39. Vous dessinerez le parcours de la recherche sur l'arbre.
Faites de même pour l'élément 6 et l'élément 11.

Exercice 41
Combien d'opérations est-il necessaire, dans le pire cas, pour trouver un élément dans un arbre binaire de recherches? Est-ce beaucoup?

Exercice 42
Donnez une implémentation de la recherche dichotomique dans un arbre binaire de recherche.

A partir d'un arbre binaire de recherche, on peut insérer une nouveau sommet d'étiquette e en utilisant *l'algorithme d'insertion*.

L'algorithme d'insertion se réalise en deux étapes :

1. La première étape consiste à exécuter une variante de l'algorithme de recherche d'un élément, où, au lieu de s'arrêter au moment où l'on trouve un sommet d'étiquette e, on continue le parcours en se déplaçant sur le fils gauche du sommet courant (si il existe).
   On note t le sommet obtenu par ce parcours.
2. la deuxième étape consiste à insérer le nouveau sommet s
   — en tant que fils droit de t si e est strictement plus grand que l'étiquette de t, et
   — en tant que fils gauche de t sinon.

L'arbre obtenu est un arbre binaire de recherche.

Par exemple, la figure 45 illustre l'insertion d'un nouveau sommet d'étiquette 9 dans un arbre binaire de recherche.

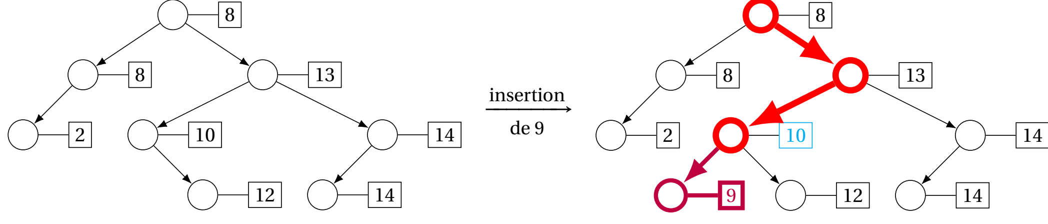


FIGURE 45 – Insertion du sommet d'étiquette 9 dans un arbre binaire de recherche.

La figure 46, quand à elle, montre l'insertion d'un sommet d'étiquette 12 dans le même arbre binaire de recherche.

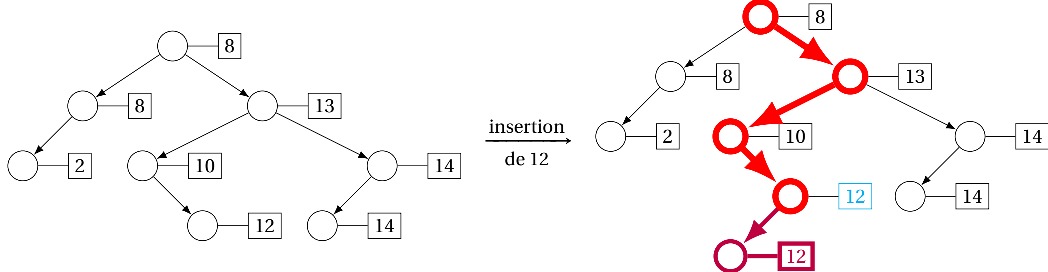


FIGURE 46 – Insertion du sommet d'étiquette 12 dans un arbre binaire de recherche.

Enfin, la figure 47, présente le cas de l'insertion d'un sommet d'étiquette 13.

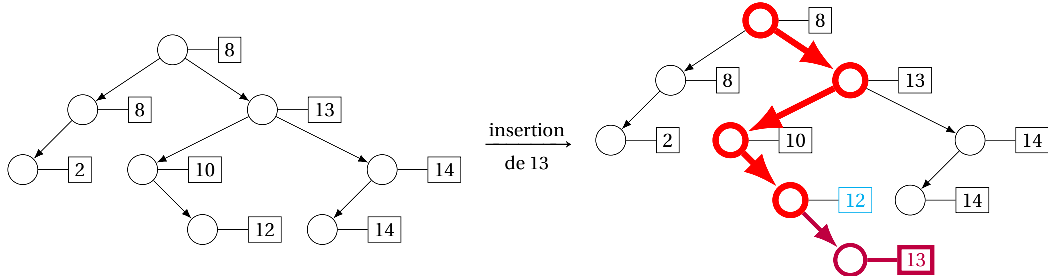


FIGURE 47 – Insertion du sommet d'étiquette 13 dans un arbre binaire de recherche.

Ainsi, pour construire un arbre binaire de recherche associé à une liste, il suffit de mettre le premier élément de la liste à la racine, puis d'insérer dans l'arbre tous les autres éléments, un à un, à l'aide de l'algorithme d'insertion.

Par exemple, l'arbre binaire de recherche obtenu à partir de la liste $l = [5, 8, 1, 2, 2, 5, 9]$ est donné à la figure 48.

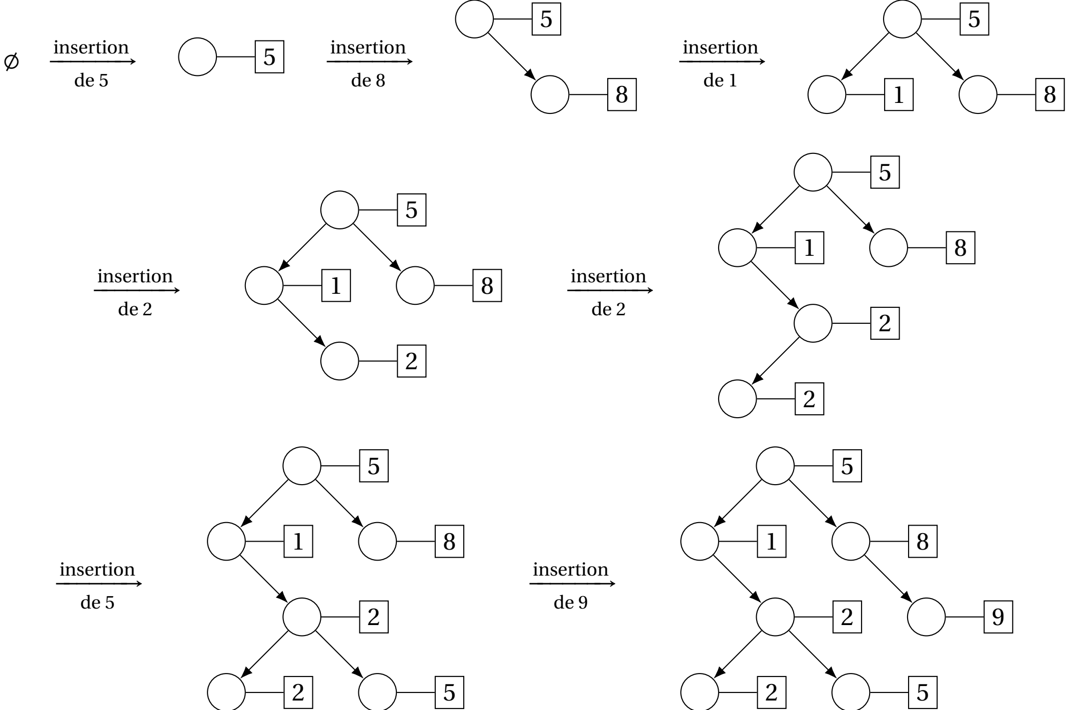


FIGURE 48 – Construction de l'arbre binaire de recherche associé à l'insertion des éléments de la liste $[5, 8, 1, 2, 2, 5, 9]$.

> **Exercice 43**
> Donnez l'arbre binaire de recherche obtenu en insérant les élément de $l = [4, 3, 5, 5, 1, 1, 2, 8, 3]$ dans un arbre initialement vide, à l'aide de l'algorithme d'insertion. Que se passe-t-il si on change l'ordre de la liste? Que se passe-t-il si on tri la liste, par ordre croissant, puis par ordre décroissant?

> **Exercice 44**
> Écrivez un programme qui à partir d'une liste d'entiers construit un arbre binaire de recherche contenant les éléments de cette liste en utilisant l'algorithme d'insertion dans les arbres binaires de recherche.

Lorsque l'on ordonne les étiquettes selon l'ordre du parcours infixe des arbres binaires de recherche, on obtient un liste triée contenant toutes les étiquettes de l'arbre.

Par exemple, le parcours infixe de l'arbre 49 donne :$[s_1, s_5, s_3, s_6, s_0, s_2, s_4]$. Si l'on remplace les sommets par leurs étiquettes, on obtient la liste triée $[1, 2, 2, 5, 5, 8, 9]$ de toutes les étiquettes de l'arbre.

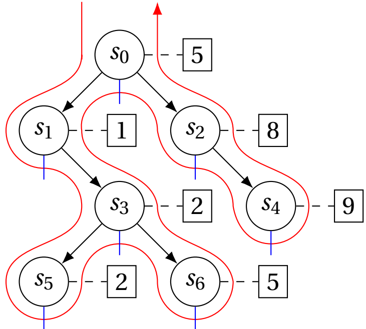


FIGURE 49 – Trier les étiquettes d'un arbre binaire de recherche à l'aide du parcours infixe.

Il est donc maintenant possible de proposer un nouvel algorithme de tri qui consiste à :

1. construire l'arbre binaire de recherche associé à la liste à triée,
2. calculer le parcours infixe de l'arbre,
3. remplacer les sommets du parcours précédent par leurs étiquettes.

Exercice 45
Appliquez l'algorithme précédent pour trier la liste $l = [3, 1, 2, 5, 4, 6, 2, 2, 8]$ dans l'ordre croissant.

Exercice 46
Écrivez en python, l'algorithme de tri utilisant les arbres binaires de recherche.

12 Les graphes

Ce cours a été écrit en utilisant le livre [2] intitulé *Graph Théory* de Diestel et le livre [1] intitulé *Introduction to Algorithms*, de Cormen et al.

12.1 Ensembles, multiensembles, listes et mots

Dans cette section nous allons rappeler les définitions d'ensemble, de multiensemble, de liste et de mot. Nous donnerons leurs notations mathématiques, ainsi que leurs implémentions en python.

Les ensembles

Un *ensemble* est une collection non ordonnée d'éléments sans répétitions. On note les ensembles en écrivant leurs éléments séparés par des virgules et entourés pas des accolades.

Par exemple, l'ensemble E constitué des éléments 1, 4 et 5 est noté par :

$$E = \{1, 4, 5\} = \{4, 1, 5\}.$$

Dans la notation, l'ordre des éléments n'est pas important car un ensemble n'est pas ordonnée.

De même comme, les ensembles sont des collections sans répétition, alors on considère que :

$$\{1, 4, 5, 1, 5\} = \{1, 4, 5\}.$$

Les multiensembles

Un multiensemble est une collection non ordonnée d'éléments avec répétition. On note les multiensemble en écrivant leurs éléments séparés par des virgules et entourés pas des doubles accolades.

Par exemple, le multiensemble M constitué de deux 1, un seul 4, trois 5 est noté par :

$$M = \{\{4,1,5,1,5,5\}\} = \{\{1,4,5,5,1,5\}\}.$$

Dans la notation, l'ordre des éléments n'est pas important car un multiensemble n'est pas ordonnée.

Ainsi, contrairement aux ensembles, on a :

$$\{\{1,4,5,1,5\}\} \neq \{\{1,4,5\}\} \qquad \text{et aussi} \qquad \{\{1,1\}\} \neq \{\{1\}\}.$$

Les mots ou les uplets

Un mot (appelé aussi uplet) est une collection ordonnée d'éléments avec répétition. On note les mots en écrivant leurs éléments, séparés par des virgules, dans l'ordre et que l'on entoure par de parenthèses.

Par exemple, le mot (ou uplet) W constitué, dans l'ordre, des éléments 4, 5, 1, 5 , 5, 1 est noté par :

$$W = (4,5,1,5,5,1).$$

Dans la notation, l'ordre des éléments est important, ainsi, $(4,5,1,5,5,1) \neq (1,1,4,5,5,5)$.

Les uplets à deux éléments sont appelés des *paires*. Par exemple, le uplet $(2,1)$ est une paire. On dit qu'un mot est sans répétition si le mot ne contient pas deux fois le même élément. Par exemple, $(1,5,4)$ est un mot sans répétition. Enfin, on appelle *permutation* un mot sans répétition dont les éléments sont en correspondance deux à deux avec $\{1,2,\cdots,n\}$. Par exemple, $(1,4,3,2)$ et $(1,2,4,3)$ sont deux permutations différentes, alors que $(1,5,3,2)$ n'est pas une permutation.

Les dictionnaires ou les associations

On appelle *dictionnaire* un ensemble de paires (k,v) où (k,v). Les paires des dictionnaires sont appelés des *associations* où k est appelé la clef et v la valeur de l'association. Un dictionnaire D vérifie la propriété que deux associations distinctes ont toujours deux clefs distinctes :

$$\forall (k_1,v_1) \in D,\ \forall (k_2,v_2) \in D, \qquad (k_1,v_1) \neq (k_2,v_2) \Longrightarrow k_1 \neq k_2.$$

Implémentation des uplets, paires, ensembles, multiensembles et permutations en python

Dans le tableau ci-dessous vous trouverez comment il est possible d'implémenter chacune des ces collections en python.

collection	Notation mathématique	structure de donnée python	code python	remarques
un uplet ou un mot	$(4,1,4,5,1)$	list	$l = [4,1,4,5,1]$	modifiable
		tuple	$l = (4,1,4,5,1)$	non modifiable
une paire	$(2,1)$	list/tuple	l = [2,1]	/
un ensemble	$\{4,1,5\}$	list/tuple trié	l = [1, 4, 5]	/
un multiensemble	$\{\{4,1,4,5,1\}\}$	list/tuple trié	l = [1, 1, 4, 4, 5]	/
une permutation	$(1,4,3,2)$	list/tuple	l = [1, 4, 3, 2]	/
un dictionnaire	$\{[1,3],[2,3],[5,4]\}$ ou bien $\{1 \to 3, 2 \to 3, 5 \to 4\}$	dict	$d = \{1:3, 2:3, 5:4\}$	/

Exercice 47
Donnez des exemples de mots, de paires, d'ensembles, de multiensembles, de permutations et de dictionnaires.
Pour chacun de vos exemples, donnez le code python équivalent.

12.2 Définition des graphes

12.2.1 Les graphes non orientés

Un graphe non orienté est une structure de données particulière que nous allons décrire sur un exemple particulier, avant de donner une définition formelle.

Un graphe non orienté est une structure constitué de sommets et d'arêtes.

Les sommets sont des éléments représentés dans la figure 50 par des numéros entourés d'un cercle. Dans le cas de la figure 50, les sommets sont v_0, v_1, v_2 et v_3.

Les arêtes, quand à elles, sont représentées dans la figure par des traits reliant les sommets. Les arêtes sont codées par un ensemble E d'éléments muni d'une *fonction d'incidence* , notée Γ, qui à toute arête e, permet d'obtenir les sommets incidents à e, c'est à dire les sommets situés aux extrémités du trait associé à l'arête e dans son dessin. Par exemple, dans la figure 50, l'arête e_2 relie les sommets V_0 et V_1.

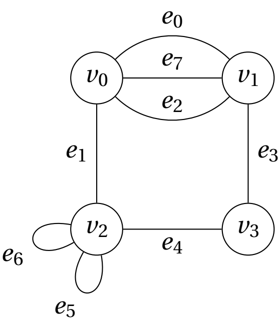


FIGURE 50 – Un graphe G non orienté.

Voici maintenant la définition formelle des graphes non orientés.

Un graphe non orienté est un 3-uplet (V, E, Γ) constitué d'un ensemble V d'éléments appelés sommets, d'un ensemble E d'éléments appelés arêtes et d'une fonction d'incidence, notée Γ qui à toute arête de E renvoi un multiensemble d'exactement 2 sommets de V.

Voici un exemple de graphe G à 4 sommets et 8 arêtes :

$$
\begin{aligned}
G &= (V, E, \Gamma),\\
V &= \{v_0, v_1, v_2, v_3\},\\
E &= \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7\},\\
\Gamma &= \left\{ \begin{matrix} e_0 : \{\{v_0, v_1\}\}, & e_1 : \{\{v_0, v_2\}\}, & e_2 : \{\{v_0, v_1\}\}, & e_3 : \{\{v_1, v_3\}\}, \\ e_4 : \{\{v_2, v_3\}\}, & e_5 : \{\{v_2, v_2\}\}, & e_6 : \{\{v_2, v_2\}\}, & e_7 : \{\{v_0, v_1\}\} \end{matrix} \right\}.
\end{aligned}
$$

Ce graphe est représentée à la figure 50.

On appelle sommet incident à une arête $e \in E$ les sommets de $\Gamma(G, e)$.

Ainsi, dans le graphe de la figure 50, les sommets incidents de e_4 sont v_2 et v_3 car $\Gamma(G, e_4) = \{\{v_2, v_3\}\}$.

On appelle *boucle* toute arête dont les sommets incidents sont identiques. Par exemple, dans le précédent graphe, les arêtes e_5 et e_6 sont des boucles car les multiensembles de $\Gamma(G, e_5)$ et $\Gamma(G, e_6)$ contiennent à chaque fois deux fois le même sommet.

Exercice 48
Donnez la définition mathématique du graphe G_1 de la figure 51.

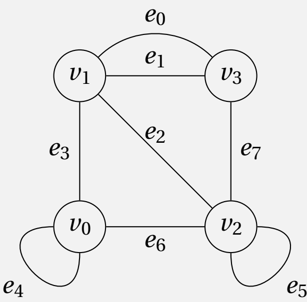


FIGURE 51 – Le graphe G_1.

Exercice 49
Dessinez le graphe G_2 défini mathématiquement par :

$$\begin{aligned}
G_2 &= (V_2, E_2, \Gamma_2)\\
V_2 &= \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}\\
E_2 &= \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, v_7\}\\
\Gamma_2 &= \left\{ \begin{array}{l} e_0 : \{\{v_0, v_2\}\},\;\; e_1 : \{\{v_1, v_2\}\},\;\; e_2 : \{\{v_0, v_1\}\},\;\; e_3 : \{\{v_4, v_5\}\}, \\ e_4 : \{\{v_4, v_5\}\},\;\; e_5 : \{\{v_1, v_1\}\},\;\; e_6 : \{\{v_1, v_1\}\},\;\; e_7 : \{\{v_4, v_4\}\} \end{array} \right\}
\end{aligned}$$

12.2.2 Les graphes orientés

Comme pour les graphes non orientés nous allons décrire les graphes orientés sur un exemple particulier avant de donner une définition formelle.

Un graphe non orienté est une structure constituée de sommets et d'arcs.

Les sommets sont des éléments représentés dans la figure 52 par des numéros entourés d'un cercle. Dans le cas de la figure 52, les sommets sont v_0, v_1, v_2 et v_3.

Les arcs, eux, sont représentés dans la figure par des flèches reliant un sommet origine à un sommet fin. Les arcs sont codés par un ensemble E d'éléments muni d'une *fonction d'incidence* , notée $\overrightarrow{\Gamma}$, qui à toute arc e, permet d'obtenir une paire de sommets, dont le premier représente l'origine de l'arc et le second la fin. Il s'agit des sommets situés aux extrémités de la flèche associée à l'arc e dans la représentation graphique du graphe. Par exemple, dans la figure 52, l'arc e_2 relie le sommet V_1 au sommet V_0 de sorte que V_1 est l'origine de e_2 et V_0 sa fin.

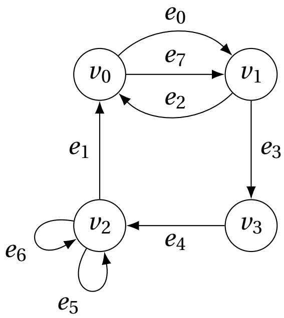


FIGURE 52 – Un graphe G orienté.

Voici maintenant la définition formelle des graphes orientés.

Un graphe orienté est un 3-uplet $(V, E, \overrightarrow{\Gamma})$ constitué d'un ensemble V d'éléments appelés sommets, d'un ensemble E d'éléments appelés arcs et d'une fonction d'incidence, notée $\overrightarrow{\Gamma}$ qui à tout arc de E renvoie une paire de 2 sommets de V.

Voici un exemple de graphe orienté G à 4 sommets et 8 arcs :

$$\begin{aligned}
G &= (V, E, \overrightarrow{\Gamma}),\\
V &= \{v_0, v_1, v_2, v_3\},\\
E &= \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7\},\\
\overrightarrow{\Gamma} &= \begin{Bmatrix} e_0 : (v_0, v_1), & e_1 : (v_2, v_0), & e_2 : (v_1, v_0), & e_3 : (v_1, v_3),\\ e_4 : (v_3, v_2), & e_5 : (v_2, v_2), & e_6 : (v_2, v_2), & e_7 : (v_0, v_1) \end{Bmatrix}.
\end{aligned}$$

Ce graphe est représentée à la figure 52.

Dans la suite de ce document on appellera *taille du graphe* le nombre de sommet de ce graphe. Ainsi, pour un graphe G donné, on le notera N ou bien $|G|$. Par exemple, la taille du graphe de la figure 52 est 4.

On appelle sommet incident à un arc $e \in E$ les sommets de $\overrightarrow{\Gamma}(G, e)$. On appelle origine d'un arc e le sommet $\overrightarrow{\Gamma}(G, e)[0]$. On appelle fin d'un arc e le sommet $\overrightarrow{\Gamma}(G, e)[1]$.

Ainsi, dans le graphe de la figure 52, les sommets incidents de e_4 sont v_2 et v_3 car $\overrightarrow{\Gamma}(G, e_4) = (v_3, v_2)$. De surcroît, v_3 est l'origine de e_4 et v_2 est sa fin.

On appelle *boucle* tout arc dont les sommets incidents sont identiques. Par exemple, dans le précédent graphe, les arcs e_5 et e_6 sont des boucles car les sommets de $\overrightarrow{\Gamma}(G, e_5)$ sont identiques. Idem pour $\overrightarrow{\Gamma}(G, e_6)$.

Exercice 50

Donnez la définition mathématique du graphe G_1 de la figure 53.

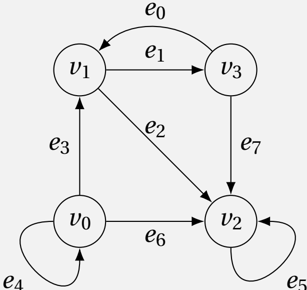


FIGURE 53 – Le graphe de l'exercice 50.

Exercice 51
Dessinez le graphe G_2 défini mathématiquement par :

$$G_2 = (V_2, E_2, \overrightarrow{\Gamma}_2)$$
$$V_2 = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}$$
$$E_2 = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, v_7\}$$
$$\overrightarrow{\Gamma}_2 = \begin{Bmatrix} e_0 : (v_0, v_2), & e_1 : (v_2, v_0), & e_2 : (v_0, v_1), & e_3 : (v_4, v_5), \\ e_4 : (v_4, v_5), & e_5 : (v_1, v_1), & e_6 : (v_1, v_1), & e_7 : (v_4, v_4) \end{Bmatrix}$$

12.2.3 Les graphes étiquetés (orientés et non orienté)

On munit les sommets et les arêtes (respectivement les arcs) d'étiquettes, c'est à dire d'une fonction qui associe à chaque sommet et à chaque arête (resp. arc) un élément appelé étiquette.

Les graphes non orientés (resp. orienté) sont alors notés par un quintuplet : (V, E, Γ, W^V, W^E) (resp. $(V, E, \overrightarrow{\Gamma}, W^V, W^E)$) où W^V et W^E sont les fonctions étiquetant respectivement les sommets et les arêtes/arcs.

12.2.4 Arêtes et arcs multiples, cas particuliers et notations abusives

Deux arêtes (resp. arcs) e_1 et e_2 sont des arêtes (resp. arcs) multiples si ils partagent la même incidences : $\Gamma(e_1) = \Gamma(e_2)$ (resp. $\overrightarrow{\Gamma}(e_1) = \overrightarrow{\Gamma}(e_2)$).

Par exemple, dans le graphe non orienté de la figure 50, les arêtes e_0, e_7 et e_2 sont des arêtes multiples car $\Gamma(e_0) = \Gamma(e_2) = \Gamma(e_7) = \{\{v_0, v_1\}\}$. De même, toujours dans ce même graphe, e_6 et e_5 sont des arêtes multiples car $\Gamma(e_6) = \Gamma(e_5) = \{\{v_2, v_2\}\}$.

De même, dans le graphe orienté de la figure 52, les arcs e_0, e_7 sont des arcs multiples car : $\overrightarrow{\Gamma}(e_0) = \overrightarrow{\Gamma}(e_7) = (v_0, v_1)$. Par contre, l'arc e_2 n'est pas un arc multiple car c'est le seul arc a avoir comme incidence $\overrightarrow{\Gamma}(e_2) = (v_1, v_0)$.

Lorsque le graphe n'a pas d'arêtes (resp. arcs) multiples, les tables Γ (resp. $\overrightarrow{\Gamma}$) et E sont abusivement fusionnés. Par exemple, la définition mathématique du graphe de la figure 54 peut, abusivement, être donnée par :

$$G = (V, E), \qquad V = \{v_0, v_1, v_2\} \qquad \text{et} \qquad E = \{\ \{\{v_0, v_1\}\},\ \{\{v_1, v_1\}\},\ \{\{v_1, v_2\}\},\ \{\{v_0, v_2\}\}\ \}.$$

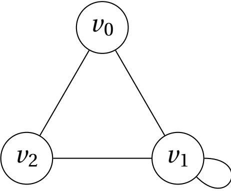


FIGURE 54 – Le graphe G.

Dans ce cas, ou quand aucune confusion ne peut être faite, les arêtes (resp. arcs) e sont alors désignées à l'aide de leur incidences $\Gamma(e)$ (resp. $\overrightarrow{\Gamma}(e)$).

Exercice 52
Donnez un exemple de graphe orienté qui possède des arcs multiples et un autre qui n'en possède pas.

Exercice 53
Donnez un exemple de graphe non orienté qui possède des arêtes multiples et un autre qui n'en possède pas.

Exercice 54
Donnez un exemple de graphe orienté qui ne possède pas d'arcs multiples, mais qui, si on retire les orientations de ses arcs, donne un graphe non orienté avec des arêtes multiples.

12.2.5 Chemins et chaînes

Dans un graphe orienté, un *chemin* de taille l est une séquence

$$s_0, a_1, s_1, a_2, s_2, \ldots, a_l, s_l$$

où s_0, s_1, ..., s_l sont des sommets et a_0, a_1, ..., a_l sont des arcs tels que, pour tout entier $i \in [\![1, l]\!]$, on ait

$$\overrightarrow{\Gamma}(a_i) = (s_{i-1}, s_i),$$

c'est à dire,

$$\begin{cases} \text{origine}(a_i) & = & s_{i-1}, \\ \text{fin}(a_i) & = & s_i. \end{cases}$$

Par exemple, dans le graphe de la figure 55, la séquence :

$$v_3, e_0, v_1, e_1, v_3, e_7, v_2, e_5, v_2, e_5, v_2$$

est un chemin dans le graphe qui part du sommets v_3, passe par les sommets v_1 et v_3 puis traverse le sommet v_2 deux fois avant de terminer sa course de nouveau en v_2.

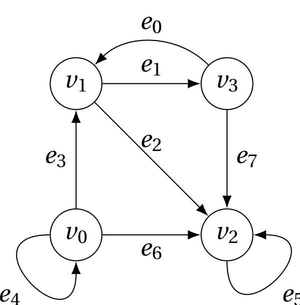


FIGURE 55 – Le graphe G_1.

Par contre, la séquence $v_1, e_2, v_2, e_6, v_0, e_4, v_0$ n'est pas un chemin du graphe car l'arc e_6 est dans le mauvais sens : $\overrightarrow{\Gamma}(e_6) \neq (v_2, v_0)$.

Dans la définition de chemin, il n'est pas possible de prendre un arc à l'envers. On souhaite donc pouvoir construire des "chemins" qui s'affranchisse de cette contrainte. C'est la définition de la chaîne.

Dans un graphe non orienté, une *chaîne* de taille l est une séquence

$$s_0, a_1, s_1, a_2, s_2, \ldots, a_l, s_l$$

où s_0, s_1, ..., s_l sont des sommets et a_0, a_1, ..., a_l sont des arcs tels que, pour tout entier $i \in [\![1, l]\!]$, on ait

$$\Gamma(a_i) = (s_{i-1}, s_i) \qquad \text{ou bien} \qquad \Gamma(a_i) = (s_i, s_{i-1}).$$

Par exemple, dans le graphe de la figure 55, la séquence :

$$v_3, e_0, v_1, e_1, v_3, e_7, v_2, e_5, v_2, e_5, v_2$$

est bien sur une chaîne dans le graphe, mais aussi la séquence $v_1, e_2, v_2, e_6, v_0, e_4, v_0$ car cette fois-ci, l'une des deux assertions suivantes est vraie : $\overrightarrow{\Gamma}(e_6) = (v_2, v_0)$ ou $\overrightarrow{\Gamma}(e_6) = (v_0, v_2)$.

Évidement, on peut étendre la définition des chaînes pour les graphes non orientés comme il suit. Dans un graphe non orienté, une *chaîne* est une séquence

$$s_0, a_1, s_1, a_2, s_2, \ldots, a_l, s_l$$

où s_0, s_1, ..., s_l sont des sommets et a_0, a_1, ..., a_l sont des arêtes telles que, pour tout entier i, on ait

$$\overrightarrow{\Gamma}(a_i) = \{\{s_{i-1}, s_i\}\}.$$

Par exemple, dans le graphe non orienté de la figure 56, la séquence :

$$v_3, e_0, v_1, e_1, v_3, e_7, v_2, e_5, v_2, e_5, v_2$$

est une chaîne dans le graphe.

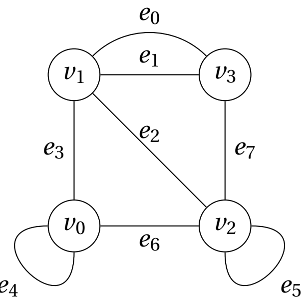


FIGURE 56 – Le graphe G_1.

De même, la séquence $v_1, e_2, v_2, e_6, v_0, e_4, v_0$ est une chaîne du graphe car, entre autres, $\overrightarrow{\Gamma}(e_6) = \{\{v_2, v_0\}\} = \{\{v_0, v_2\}\}$.

Exercice 55

— Donnez des exemples de chemins et de chaînes dans le graphe de la figure 57.

— Proposez, des exemples de séquences qui ne sont, ni des chemins, ni des chaînes.

— Donnez des exemples de chaînes qui ne sont pas des chemins.

— Peut-on trouver des chemins qui ne sont pas des chaînes? Justifier votre réponse.

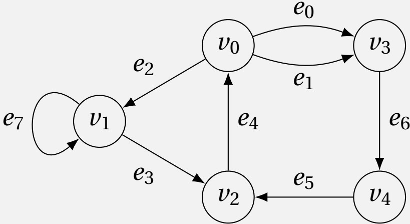


FIGURE 57 – Le graphe de l'exercice 55.

12.2.6 Accessibilité, connexité, forte conexité, composantes connexes et composantes fortement connexes

On dit que le sommet s_1 est *accessible* depuis le sommet s_0 s'il existe un chemin (resp. une chaîne) qui relie s_0 à s_1 dans le graphe orienté (resp. non orienté).

Par exemple, dans le graphe orienté de la figure 58, le sommet v_2 est accessible depuis le sommet v_0. Par contre, les sommets v_3 et v_4 ne sont pas accessibles depuis v_0.

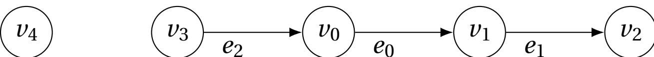


FIGURE 58 – Le sommets v_2 est accessible depuis v_0. Les sommets v_3 et v_4 ne sont pas accessibles depuis v_0.

Deux sommets s_0 et s_1 sont *connexes* s'il existe une chaîne qui relie le sommet s_0 à s_1.

Par exemple, dans le graphe de la figure 59, les sommets v_4 et v_6 sont connexes car la chaîne

$$v_4, e_0, v_3, e_5, v_5, e_6, v_6$$

relie le sommet v_4 au sommet v_6. Par contre, les sommets v_0 et v_4 ne sont pas connexes.

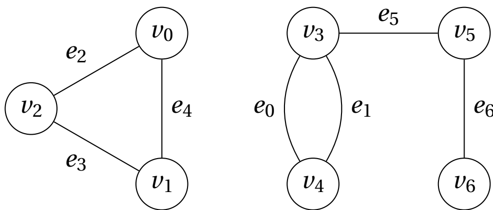


FIGURE 59 – Les sommets v_4 et v_6 sont connexes.

En fait, la notion de connexité ne dépends pas des arcs, ainsi, si l'on ajoute des orientations au graphe de la figure 59 pour obtenir celui de la figure 60, alors, les sommets v_4 et v_6 restent toujours connexes car la chaîne $v_4, e_0, v_3, e_5, v_5, e_6, v_6$ relie toujours le sommet v_4 au sommet v_6. On rappelle que les chaînes, contrairement aux chemins, ne prennent pas en compte les orientations des arcs.

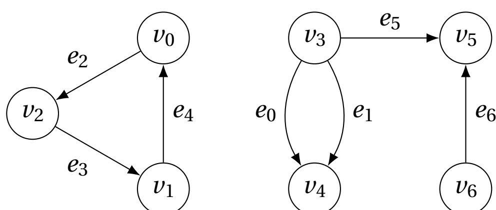


FIGURE 60 – Les sommets v_4 et v_6 sont connexes.

Un ensemble E est une *composante connexe* de G s'il vérifie les deux propriétés suivantes :

1. tous les éléments de E sont connexes deux à deux dans G,
2. E est un ensemble maximal, c'est à dire que, pour tout sommet $s_0 \notin E$ et pour tout sommet $s_1 \in E$ les sommets s_0 et s_1 ne sont pas connexes dans G.

Ainsi, toujours dans le graphe de la figure 60, l'ensemble $E = \{v_3, v_4, v_5\}$ n'est pas une composante connexe car E n'est pas maximal, alors que l'ensemble $F = \{v_3, v_4, v_5, v_6\}$ est une composante connexe du graphe car il est connexe et maximal. Ainsi, dans ce même graphe, il y a deux composantes connexes qui sont $\{v_0, v_1, v_2\}$ et $\{v_3, v_4, v_5, v_6\}$.

Exercice 56
Dans le graphe de la figure 61, donnez deux sommets connexes et deux sommets qui ne sont pas connexes.
Enfin, donnez toutes les composantes connexes du graphe.

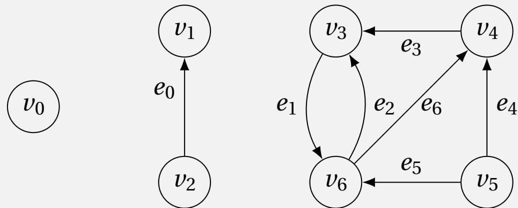


FIGURE 61 – Le graphe des exercices 56 et 58.

On dit qu'un graphe est *connexe* s'il ne contient qu'une seule composante connexe.

Exercice 57
Donnez un exemple de graphe connexe et un exemple de graphe qui n'est pas connexe.

Dans un graphe orienté, deux sommets s_0 et s_1 sont *fortement connexes* s'il existe à la fois un chemin reliant s_0 à s_1, mais aussi un chemin reliant s_1 à s_0.

Par exemple, dans la figure 60, les sommets v_4 et v_6 ne sont pas fortement connexes car il n'existe pas de chemin reliant v_4 à v_6. Par contre les sommets v_1 et v_2 sont fortement connexes car il existe un chemin reliant v_2 à v_1 qui est v_2, e_3, v_1 et il existe aussi un chemin qui relie v_1 à v_2 qui est v_1, e_4, v_0, e_2, v_2.

Un ensemble E est une *composante fortement connexe* de G si il vérifie les deux propriétés suivantes :

1. tous les éléments de E sont fortement connexes deux à deux dans G,
2. E est un ensemble maximale, c'est à dire que, pour tout sommet $s_0 \notin E$ et tout sommet $s_1 \in E$ s_0 et s_1 ne sont pas fortement connexes.

Dans le graphe de la figure 60, il y a 5 composantes fortement connexes qui sont : $\{v_0, v_1, v_2\}$, $\{v_3\}$, $\{v_4\}$, $\{v_5\}$ et $\{v_6\}$.

Exercice 58
Reprenez le graphe de l'exercice 56 et donnez deux sommets fortement connexes et deux sommets qui ne sont pas fortement connexes, mais tout de même connexes.
Enfin, donnez toutes les composantes fortement connexes du graphe.

On dit qu'un graphe est *fortement connexe* si il ne contient qu'une seule composante fortement connexe.

Exercice 59

1. Donnez un exemple de graphe fortement connexe et un exemple de graphe qui n'est pas fortement connexe.
2. Donnez un exemple de graphe connexe qui n'est pas fortement connexe.

12.2.7 Cycles, circuits et arbres

Un cycle élémentaire est une chaîne de taille $l \geq 1$ qui commence et termine par le même sommet et qui ne passe (à l'exception de ce dernier) qu'une et une fois par sommet et qu'un et une fois par arête.

Par exemple, dans le graphe de la figure 62, la chaîne $v_0, e_3, v_1, e_4, v_2, e_1, v_3, e_0, v_0$ est un cycle élémentaire.

Par contre, la chaîne v_0, e_3, v_1, e_3, v_0 n'est pas un cycle élémentaire car il passe deux fois par e_3. De même, la chaîne $v_1, e_3, v_0, e_0, v_3, e_2, v_1, e_5, v_4, e_6, v_2, e_4, v_1$ n'est pas un cycle élémentaire car il passe deux fois par le sommet v_1.

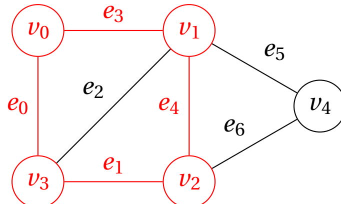


FIGURE 62 – La chaîne $v_0, e_3, v_1, e_4, v_2, e_1, v_3, e_0, v_0$ est un cycle élémentaire.

Exercice 60
Donnez tous les cycles élémentaires du graphe de la figure 62.

Un circuit élémentaire est une chemin de taille $l \geq 1$ qui commence et termine par le même sommet et qui ne passe (à l'exception de ce dernier) qu'une et une fois par sommet qui compose le chemin (et donc qu'une et une fois par arcs).

Par exemple, dans le graphe de la figure 63, le circuit $v_0, e_3, v_1, e_4, v_2, e_1, v_3, e_0, v_0$ est un circuit élémentaire.

Par contre, le circuit $v_1, e_2, v_3, e_0, v_0, e_3, v_1, e_4, v_2, e_6, v_4, e_5, v_1$ n'est pas un cycle élémentaire car il passe deux fois par le sommet v_1.

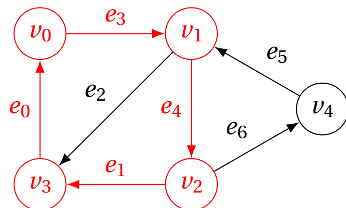


FIGURE 63 – Le circuit $v_0, e_3, v_1, e_4, v_2, e_1, v_3, e_0, v_0$ est un chemin élémentaire.

Exercice 61
Donnez tous les circuit élémentaires du graphe de la figure 63.

Un *arbre* est un graphe connexe sans cycle.

Par exemple, le graphe de la figure 64 est un arbre car il est connexe et sans cycle.

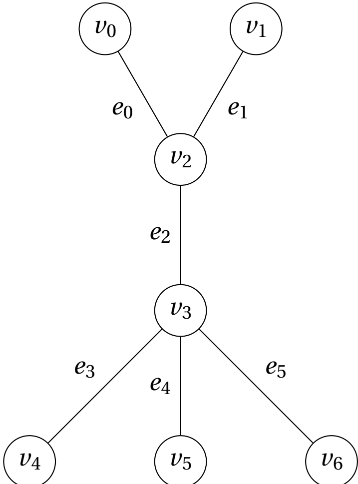
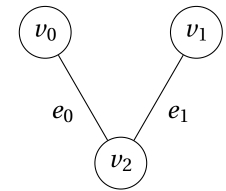


FIGURE 64 – Un exemple d'arbre.

A contrario, les graphes de la figure 62 et 63 ne sont pas des arbres car ils possèdent tous les deux des cycles élémentaires.

De même, le graphe de la figure 65 n'est pas un arbre car il n'est pas connexe.



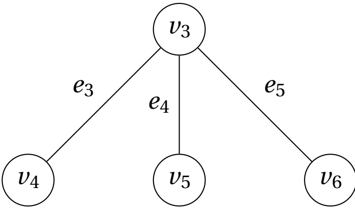


FIGURE 65 – Un graphe qui n'est pas un arbre car il a deux composantes connexes.

Exercice 62
Construisez des exempels de graphes qui sont des arbres et d'autres qui n'en sont pas.

Exercice 63: (difficile)
Si on note par n le nombre de sommets d'un graphe, montrez que dans un arbre il y a toujours $n-1$ arêtes.

12.2.8 Degré d'un sommet et degré total d'un graphe

Le *degré* d'un sommet est le nombre d'arêtes (resp. arcs) incidentes à ce sommet où les boucles comptent doubles. On notera par $d(s)$ le degré du sommet s.

Le *degré total* d'un graphe, que l'on note $d(G)$, est la somme des degrés de tout les sommets du graphe :

$$d(G) = \sum_{s \in V} d(s).$$

Par exemple, dans la figure 66, le sommet v_0 a pour degré 0, le sommet v_2 a pour degrés 3, le sommets v_3 a pour degré 4 et le sommet v_4 a pour degré 2.

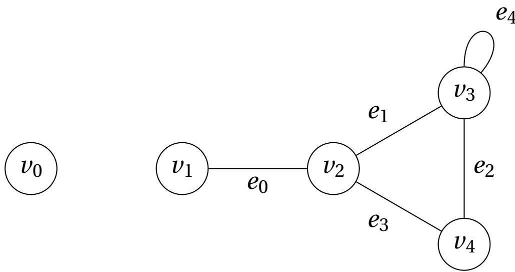


FIGURE 66 – Un graphe non orienté dont le bilan des degrés du graphe est $\{\{4, 3, 2, 1, 0\}\}$.

Dans un graphe orienté, le *degré entrant* d'un sommet s est le nombre d'arcs ayant s pour origine. On notera par $d^+(s)$ le degré entrant du sommet s.

Le *degré sortant* d'un sommet s est le nombre d'arcs ayant s pour fin. On notera par $d^-(s)$ le degré sortant du sommet s.

Par exemple, dans la figure 67, les dégrés entrants et sortant et les degrés sont :

$$\begin{array}{lll}
d^-(v_0) = 0 & d^+(v_0) = 0 & d(v_0) = 0 \\
d^-(v_1) = 1 & d^+(v_1) = 0 & d(v_1) = 1 \\
d^-(v_2) = 2 & d^+(v_2) = 1 & d(v_2) = 3 \\
d^-(v_3) = 2 & d^+(v_3) = 2 & d(v_3) = 4 \\
d^-(v_4) = 0 & d^+(v_3) = 2 & d(v_4) = 2
\end{array}$$

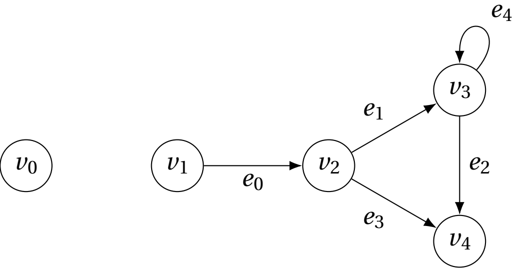


FIGURE 67 – Un graphe orienté dont la table des degrés est $[4, 3, 2, 1, 0]$.

Évidement, dans un graphe orienté, le degré d'un sommet est égal à son degré entrant plus son degré sortant :

$$d(s) = d^-(s) + d^+(s).$$

Lemme 1. *Le degré total d'un graphe est égal à deux fois son nombre d'arêtes (resp. arcs) :*

$$d(G) = 2 \times |E|$$

Démonstration. Chaque arête (resp. arc) compte pour deux dans le degrés total du graphe. □

Par exemple, dans le graphe de la figure 67 il y a 5 arcs, et donc, le degré total du graphe est de $2 \times 5 = 10$.

Exercice 64
Un graphe simple G a 15 arêtes, 3 sommets de degré 4, les autres étant de degré 3. Quel est le nombre de sommets du graphe G?

Exercice 65
Est-il possible qu'un groupe de 9 personnes soit tel que chacun soit l'ami de 5 personnes exactement du groupe ?

Exercice 66
Montrer que dans tout graphe, il y a un nombre pair de sommets de degré impair.

Exercice 67: (difficile)
Dans un graphe connexe un isthme est une arête dont la suppression augmente le nombre de composantes connexes. Montrer qu'un graphe dont tous les sommets sont de degré pair ne possède pas d'isthme.

12.3 Implémentation des graphes

Il existe plusieurs façons d'implémenter les graphes dans un programme. Dans cette section, nous allons présenter 4 structures de données usuelles : les matrices d'adjacence, les listes d'arcs ou arêtes adjacents, les listes d'adjacence (de sommets adjacents) et les matrices d'incidence.

Chacune des ces structures possèdent des avantages et des inconvénients. Lorsque l'on souhaite construire des algorithmes en parcourant les graphes de sommet en sommet, les listes d'incidences (pour les graphes à arêtes multiples) et les listes d'adjacence (pour les graphes sans arêtes multiples) sont préférées.

Lorsque l'on souhaite obtenir des propriétés algébriques sur les graphes, on préfère utiliser les matrices d'adjacences et les matrices d'incidence.

Nous allons maintenant présenter ces 4 structures de données avant de résumer leurs avantages et inconvénients.

12.3.1 Matrice d'adjacence

On peut implémenter les graphes orientés à l'aide d'un tableau A d'entiers appelé *matrice d'adjacence* où l'entrée $A[i][j]$ du tableau (entrée située à la ligne i et à la colonne j) correspond au nombre d'arcs reliant le sommet i au sommet j.

Par exemple, le graphe orienté de la figure 68 est implémenté à l'aide de la matrice d'adjacence A suivante :

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 \end{pmatrix}. \tag{2}$$

Dans cette matrice, l'entrée $T[3][1]$ contient l'entier 2 ce qui signifie qu'il y a 2 arcs reliant le sommet 3 au sommet 1 comme on peut l'observer dans la figure 68. Par contre, comme $T[1][6] = 0$, il n'y a pas d'arc qui relie le sommet 1 au sommet 6.

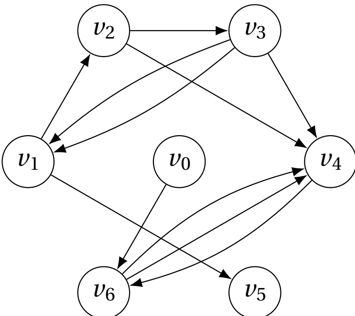


FIGURE 68 – Le graphe orienté associé à la matrice A de l'équation 2.

Cette implémentation, très compacte, ne permet pas de différencier deux arcs ayant les mêmes incidences, c'est pourquoi les arcs ne sont pas numérotés.

On peut implémenter les graphes non orientés de la même manière. On obtient alors une matrice qui admet un axe de symétrie qui est l'axe Nord-ouest/Sud-est. Par exemple, la matrice :

$$B = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}. \tag{3}$$

peut représenter aussi bien le graphe orienté de la figure 69 (situé à gauche) que le graphe non orienté de cette même figure (situé à droite). Dans cette matrice, on peut observer l'axe de symétrie Nord-ouest/Sud-est qui sépare les entrées écrites à l'encre noire de leurs symétries écrites à l'encre grise.

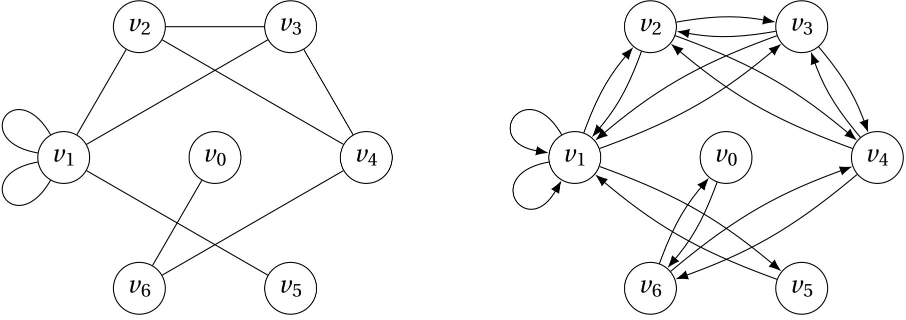


FIGURE 69 – Deux graphes (orienté et non orienté) ayant la même matrice d'adjacence qui est définie à l'équation 3.

Exercice 68
Soit la matrice C suivante :

$$C = \begin{pmatrix} 1 & 2 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Est-ce la matrice d'adjacence d'un graphe orienté ou d'un graphe non orienté ? Dessinez les graphes orientés et éventuellement non orientés qui lui sont associés.

Exercice 69
Soit la matrice D suivante :

$$D = \begin{pmatrix} 1 & 2 & 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

Est-ce la matrice d'adjacence d'un graphe orienté ou d'un graphe non orienté ? Dessinez le graphe orienté et éventuellement non orienté qui lui sont associés.

Exercice 70
Donnez la matrice d'adjacence associée au graphe orienté de la figure 70.

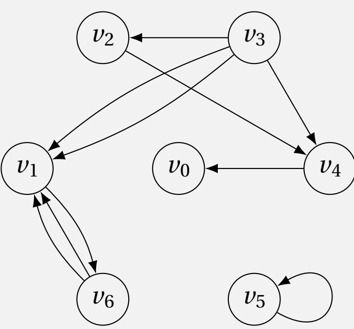


FIGURE 70 – Le graphe orienté de l'exercice 70.

Exercice 71
Donnez la matrice d'adjacence associée au graphe orienté de la figure 71.

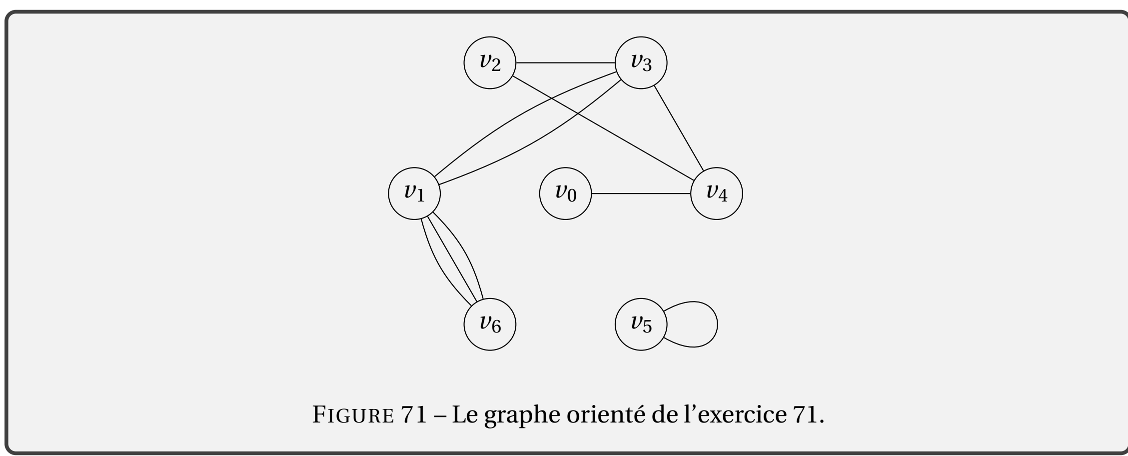


FIGURE 71 – Le graphe orienté de l'exercice 71.

Enfin, lorsque le graphe ne contient pas d'arêtes (ou d'arcs) multiples, alors les réels présents sont utilisés pour coder les étiquettes du graphe. Dans ce cas, les arcs d'étiquettes 0 ne sont pas représentés dans le graphe.

Ainsi, si l'on interprète la matrice suivante :

$$E = \begin{pmatrix} 0 & 0.3 & 0 & 0 \\ 4.1 & 0 & 0 & 0 \\ 0.3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{4}$$

comme un graphe étiqueté à arcs simples, alors on obtient le graphe de la figure 72.

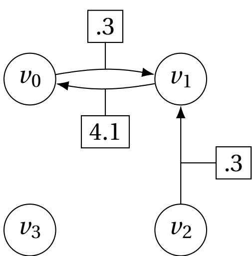


FIGURE 72 – Le graphe étiqueté à arcs simples associé à la matrice d'adjacence E de l'équation 4.

Exercice 72

Donnez la matrice d'adjacence du graphe étiqueté de la figure 73.

v_0 v_1 9. 5.1 8. v_3 v_2 8.

FIGURE 73 – Le graphe de l'exercice 72.

Exercice 73
Dessinez le graphe non orienté simple et étiqueté défini par la matrice d'adjacence suivante :

$$F = \begin{pmatrix} 2.1 & 0 & 1.2 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2.1 \\ 2.1 & 0 & 4 & 0 \end{pmatrix}.$$

12.3.2 Liste d'arcs adjacents et liste d'arêtes adjacentes

La liste d'arcs (resp. arêtes) adjacents est un dictionnaire D_I dont les clefs sont les sommets du graphe et les valeurs des listes d'arcs (resp. arêtes) telles que, pour tout sommet s du graphe, $D_I[s]$ est la liste des arcs (resp. arêtes) sortantes de s :

$$D_I[s] = [e \in E | \, \text{origine}(e) = s].$$

. La liste d'arcs (resp. arêtes) adjacents est toujours accompagné de la table $\overrightarrow{\Gamma}$ (resp. Γ) qui permet de connaître les sommets incidents d'un arc (resp. d'une arête).

Par exemple, la liste d'arcs adjacents et la fonction d'incidence suivante,

$$\begin{cases} D_I = \{\ v_0 : [e_0],\ v_1 : [e_1, e_2],\ v_2 : [e_5, e_6],\ v_3 : [e_3, e_4, e_7],\ v_4 : [e_{10}],\ v_5 : [],\ v_6 : [e_8, e_9]\ \}, \\ \overrightarrow{\Gamma} = \begin{Bmatrix} e_0 : (v_0, v_6),\ e_1 : (v_1, v_2),\ e_2 : (v_1, v_5),\ e_3 : (v_3, v_1),\ e_4 : (v_3, v_1),\ e_5 : (v_2, v_3), \\ e_6 : (v_2, v_4),\ e_7 : (v_3, v_4),\ e_8 : (v_6, v_4),\ e_9 : (v_6, v_4),\ e_{10} : (v_4, v_6) \end{Bmatrix} \end{cases} \tag{5}$$

sont représentées par le graphe de la figure 74. Dans ce graphe, comme $D_I[v_1] = [e_1, e_2]$, le sommet v_1 est l'origine des arcs e_1 et e_2. De même, comme $D_I[v_1] = []$, le sommet v_5 n'est l'origine d'aucun arc.

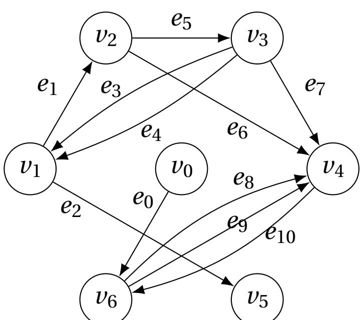


FIGURE 74 – Le graphe orienté associé à la liste d'arcs adjacents D_I de l'équation 5.

La version non orienté de ce graphe donne lieu à la liste d'arêtes adjacentes suivante :

$$\begin{cases} D_I = \begin{Bmatrix} v_0 : [e_0],\ v_1 : [e_1, e_2, e_3, e_4],\ v_2 : [e_1, e_5, e_6],\ v_3 : [e_3, e_4, e_5, e_7], \\ v_4 : [e_6, e_7, e_8, e_9, e_{10}],\ v_5 : [e_2],\ v_6 : [e_0, e_8, e_9, e_{10}] \end{Bmatrix}, \\ \\ \overrightarrow{\Gamma} = \begin{Bmatrix} e_0 : \{\{v_0, v_6\}\},\ e_1 : \{\{v_1, v_2\}\},\ e_2 : \{\{v_1, v_5\}\},\ e_3 : \{\{v_3, v_1\}\},\ e_4 : \{\{v_3, v_1\}\},\ e_5 : \{\{v_2, v_3\}\}, \\ e_6 : \{\{v_2, v_4\}\},\ e_7 : \{\{v_3, v_4\}\},\ e_8 : \{\{v_6, v_4\}\},\ e_9 : \{\{v_6, v_4\}\},\ e_{10} : \{\{v_4, v_6\}\} \end{Bmatrix} \end{cases} \tag{6}$$

qui est représentée par le graphe de la figure 75.

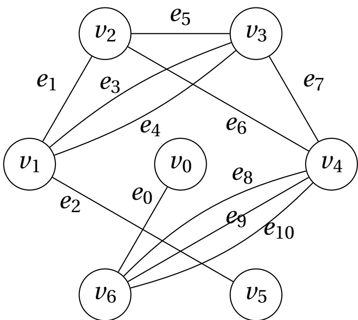


FIGURE 75 – Le graphe non orienté associé à la liste d'arêtes adjacentes D_I de l'équation 6.

12.3.3 Liste d'adjacence (liste de sommets adjacents)

La liste d'adjacence est similaire à la liste d'arcs/arêtes adjacents. Elle est utilisée quand le graphe n'a pas d'arc (resp. arête) multiple et qu'il n'est donc pas nécessaire de les numéroter.

La *liste d'adjacence* ou *liste de sommets adjacents* est un dictionnaire Adj dont les clefs sont les sommets du graphes et les valeurs des listes d'arcs (resp. arêtes) telles que, pour tout sommet s du graphe, Adj[s] est

— pour un graphe orienté la liste des fins des arcs du graphe ayant pour origine s :

$$\mathrm{Adj}[s] = [t \in V \mid \exists e \in E,\ \overrightarrow{\Gamma}(e) = (s,t)].$$

— pour un graphe non orienté la liste des sommets t des arêtes ayant pour extrémités $\{\{s,t\}\}$:

$$\mathrm{Adj}[s] = [t \in V \mid \exists e \in E,\ \Gamma(e) = \{\{s,t\}\}].$$

Contrairement aux listes d'arcs/arêtes adjacents, cette structure n'est pas accompagnée de la table $\overrightarrow{\Gamma}$ (resp. Γ) car les arcs et arêtes ne sont pas numérotés.

Par exemple, la liste d'adjacence suivante :

$$\mathrm{Adj} = \{\ v_0 : [v_6],\ \ v_1 : [v_2, v_5],\ \ v_2 : [v_3, v_4],\ \ v_3 : [v_1, v_4],\ \ v_4 : [v_6],\ \ v_5 : [],\ \ v_6 : [v_4]\ \} \tag{7}$$

est représentée par le graphe orienté de la figure 76. Dans ce graphe, comme Adj[v_1] = $[v_2, v_5]$ alors v_1 est l'origine des arcs (v_1, v_2) et (v_1, v_5).

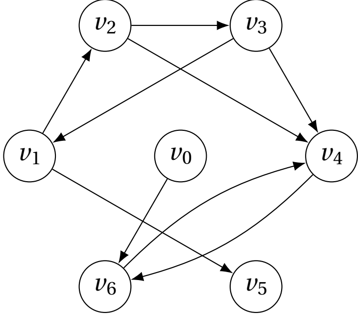


FIGURE 76 – Le graphe orienté associé à la liste d'adjacence Adj de l'équation 7.

La version non orienté de ce graphe donne lieu à la liste d'incidence suivante :

$$\mathrm{Adj} = \{\ v_0 : [v_6],\ \ v_1 : [v_2, v_3, v_5],\ \ v_2 : [v_1, v_3, v_4],\ \ v_3 : [v_1, v_2, v_4],\ \ v_4 : [v_2, v_3, v_6],\ \ v_5 : [v_1],\ \ v_6 : [v_0, v_4]\ \} \tag{8}$$

qui est représentée par le graphe de la figure 77.

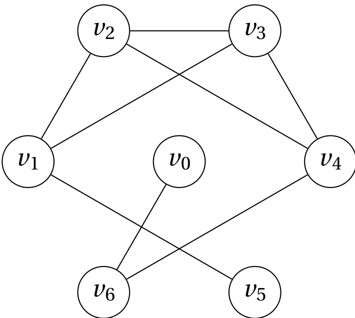


FIGURE 77 – Le graphe orienté associé à la liste d'adjacence Adj de l'équation 8.

12.3.4 Matrice d'incidence

On peut implémenter les graphes orientés sans boucle à l'aide d'un tableau M d'entiers appelé *matrice d'incidence* de hauteur n et de largeur m où n et m sont respectivement le nombre de sommets et le nombre d'arcs (resp. arête) du graphe.

Dans cette matrice, l'entrée $M[i][j]$ du tableau (entrée située à la ligne i et à la colonne j) vaut :

$$M[i][j] = \begin{cases} -1 & \text{si } v_i \text{ est l'origine de l'arc } e_j; \\ +1 & \text{si } v_i \text{ est la fin de l'arc } e_j. \end{cases}$$

Par exemple, le graphe orienté de la figure 78 est implémenté à l'aide de la matrice d'incidence M suivante :

$$M = \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & +1 & +1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & +1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & +1 & +1 & +1 & +1 & -1 \\ 0 & 0 & +1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ +1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & +1 \end{pmatrix}. \quad (9)$$

Dans cette matrice, l'entrée $M[1][2]$ contient l'entier -1 ce qui signifie que le sommet v_1 est l'origine de l'arc e_2. Comme l'entrée $M[5][2]$ vaut +1, le sommet v_5 est la fin de l'arc e_2. Par contre, comme $M[0][2] = 0$ le sommet v_0 n'est pas incident à l'arc e_2. On peut observer tout cela dans la figure 78.

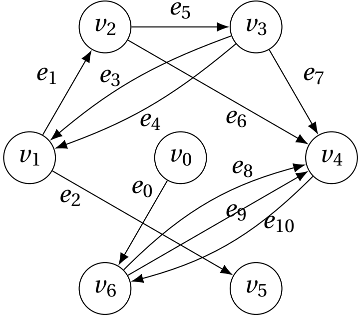


FIGURE 78 – Le graphe orienté associé à la matrice d'incidence M de l'équation 9.

Généralement, on n'utilise pas les matrice d'incidence pour implémenter des graphes avec des boucles ou des graphe non orienté, car on perd certaines propriétés algébriques de la matrice d'incidence.

Cependant, on peut toujours adapter cette structure pour implémenter un graphe quelconque. Pour un graphe non orienté, il suffit que de remplacer les valeurs -1 par 1. De même, pour une boucle e_j de sommet v_i, il suffit d'imposer la valeur 2 à l'entrée $M[i][j]$.

Exercice 74
Dessinez le graphe orienté associé à la matrice d'incidence suivante :

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

Exercice 75
Donnez la matrice d'incidence du graphe de la figure 79.

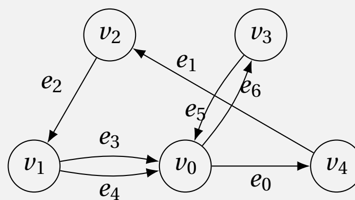


FIGURE 79 – Le graphe orienté de l'exercice 75.

12.3.5 Avantages et inconvénients des différentes structures de données de graphe

12.4 Parcours en Largeur et parcours en profondeur

12.4.1 Parcours en Largeur

```
PL(G,s)
  pour chaque sommet u de V[G] \ {s} faire
      couleur[u] <– BLANC
      d[u] <– infini
      pere[u] <– nil
  couleur[s] <– GRIS
  d[s] <– 0
  pere[s] <– nil
  Enfiler(F, s)
  tant que non vide(F) faire
      u <– tete(F)
      pour chaque v de Adj(u) faire
          si couleur[v] = BLANC
              alors couleur[v] <– GRIS
                    d[v] <– d[u] + 1
                    pere[v] <– u
                    Enfiler(F, v)
      Defiler(F)
      couleur[u] <– NOIR
```

12.4.2 Parcours en profondeur

```
Visiter_PP(u)
  couleur[u] <– GRIS
  d[u] <– temps <– temps + 1
  pour chaque v de Adj[u] faire
      si couleur[v] = BLANC
          alors pere[v] <– u
                Visiter_PP(v)
  couleur[u] <– NOIR
  f[u] <– temps <– temps + 1

PP(G)
  pour chaque sommet u de V faire
      couleur[u] <– BLANC
      pere[u] <– nil
  temps <– 0
  pour chaque sommet u de V faire
      si couleur[u] = BLANC
          alors Visiter_PP(u)
```

12.5 Algorithmes de plus court chemin

12.6 Dijkstra

```
Dijkstra(G,w,s)
  pour chaque sommet v de V(G) faire
      d[v] <– infini
      pere[v] <– nil
      couleur[v] <– BLANC
  d[s] <– 0
  F <– FILE_PRIORITE(V(G), d)
  tant que F non vide faire
      pivot <– EXTRAIRE_MIN(F)
      couleur[pivot] <– GRIS
      pour tout arc e = (pivot, v) arc sortant de pivot faire
          si couleur(v) = BLANC et d[v] > d[pivot] + w(e)
              alors d[v] <– d[pivot] + w(e)
                    pere[v] <– pivot
      couleur[pivot] <– NOIR
```

12.6.1 Bellman

12.7 Tri topologique et calcul de composantes fortement connexes

12.7.1 Tri topologique

```
TRI_TOPOLOGIQUE(G)
  1/ appeler PP(G) pour calculer les temps de fin de traitement f[v]
     pour chaque sommet v
  2/ chaque fois que le traitement d'un sommet s'achève, l'insérer
     au début d'une liste chaînée
  3/ retourner la liste chaînée des sommets
```

12.7.2 Composantes fortement connexes

```
CFC(G)
  Exécuter PP(G) et trier les sommets selon un ordre décroissant de f[]
  Calculer t(G), le graphe transposé de G, en renversant chaque arc de G
  Exécuter PP(t(G)) en respectant l'ordre obtenu en 1
  Retourner les arborescences obtenues comme composantes fortement connexes de G
```

13 Références

[1] Thomas H. CORMEN et al. *Introduction to Algorithms*. 3nd. The MIT Press, 2009. ISBN : 978-0-262-03384-8.

[2] Reinhard DIESTEL. *Graph Theory*. 5nd. T. 173. Graduate Texts in Mathematics. Springer-Verlag, Heidelberg, 2010. ISBN : 978-3-662-53621-6.

A Le langage de programmation Python

A.1 Écrire et exécuter un programme écrit en Python

Un programme écrit en python est un texte contenant des instructions écrite de haut en bas. Chaque ligne du texte correspond à une instruction. Sauf cas particulier, les instructions DOIVENT être alignées verticalement à gauche.

Voici un exemple de code écrit en python :

```
a = 5 # <-- Première instruction
a = a / 2 # <-- Deuxième instruction
print( a ) # <-- Troisième instruction
```

Voici un exemple de code écrit en python qui n'est pas valide car les instructions ne sont pas alignée verticalement à gauche :

```
a = 5 # <-- Première instruction
    a = a / 2 # <-- Deuxième instruction
print( a ) # <-- Troisième instruction
```

Sauf cas particulier, l'ordinateur exécute les instructions d'un programme de haut en bas et de gauche à droite.

Même si les instructions ressembles à des formules mathématiques, ils n'ont pas la même signification !

Lorsque une instruction est exécutée, l'ordinateur n'affiche généralement rien à l'écran. L'absence d'affichage ne veux pas dire que l'ordinateur ne fait rien.

Ainsi, si vous exécutez les instructions suivantes :

```
3 + 2
5 / 7
2 - 6
```

L'ordinateur réalise les toutes les instructions écrites (ici ce sont des calculs arithmétiques) mais n'affiche rien à l'écran.

Si vous voulez afficher le résultat d'une instruction, il faut écrire

```
print( INSTRUCTION )
```

Par exemple, le programme

```
3 + 2
print( 5 - 7 )
2 / 6
```

affichera -2.

A.2 Les variables et les affectations

A.2.1 Les variables

Nous avons vu que le rôle de l'ordinateur était de manipuler la mémoire. En python, la manipulation de la mémoire se fait à l'aide des variables.

Une *variable* est le nom que l'on donne à un petit espace mémoire. Nous rappelons qu'en python un petit espace mémoire représente 32 ou 64 bits seulement.

On peut créer autant de variable que l'on souhaite et leur donner le nom que l'on veut à condition qu'il ne comporte que des caractères alphanumérique.

Pour créer une variable de nom `NOM_DE_LA_VARIABLE`, il faut écrire l'instruction suivante :

```
NOM_DE_LA_VARIABLE = INSTRUCTION
```

où `INSTRUCTION` est une autre instruction python.

Lorsque l'on écrit ce code, l'ordinateur alloue un petit espace mémoire (32 ou 64 bits), le réserve pour le programme et lui donne comme nom `NOM_DE_LA_VARIABLE`. Ensuite, il exécuté l'instruction `INSTRUCTION` et le résultat de cette instruction est enregistré dans l'espace mémoire associé à `NOM_DE_LA_VARIABLE`.

Par exemple, le programme

```
tmp = 5
```

alloue un petit espace mémoire et lui donne le nom `tmp`. Ensuite, il évalue 5, le résultat est la valeur 5 et il l'enregistre dans la mémoire associée à `tmp`.

Attention, l'affectation n'a pas le même sens que le signe = en mathématique. Ainsi,

```
5 = a
```

n'est pas une instruction valide! Et n'a aucune signification en python!

Une fois la variable `NOM_DE_LA_VARIABLE` crée, l'instruction

```
NOM_DE_LA_VARIABLE = INSTRUCTION
```

modifie le contenu de `NOM_DE_LA_VARIABLE` en le remplaçant par le résultat de l'instruction `INSTRUCTION`.

Ainsi, le programme

```
a = 3
a = 5
```

alloue un espace mémoire pour a et stock dans cet espace la valeur 5, puis remplace cette valeur par 3.

A.2.2 Afficher le contenu d'une variable

En python il est possible d'afficher le contenu d'une variable sur le terminal. Il suffit d'utiliser l'instruction :

```
print( NOM_DE_LA_VARIABLE )
```

Par exemple, le programme :

```
a = 4
print( a )
```

alloue un petit espace mémoire et l'appelle `a`, met ensuite 4 dans l'espace associé à a et enfin, affiche sur le terminal le contenu de la variable a. Il apparaît alors à l'écran le chiffre 4.

A.2.3 Copier le contenu de la mémoire

A chaque fois que l'on utilise une variable, l'ordinateur l'évalue en récupérant le contenu de l'espace mémoire qui lui est associé. Ainsi, pour copier le contenu de l'espace mémoire, il suffit d'utiliser l'affectation vu précédemment de la façon suivante :

```
NOM_DE_LA_VARIABLE_DE_DESTINATION = NOM_DE_LA_VARIABLE_A_COPIER
```

Par exemple, dans le programme suivant :

```
b = 5
a = 1
b = a
print( b )
```

lorsque l'ordinateur exécute la troisième instruction, l'ordinateur alloue un espace mémoire pour la variable b et rempli cette espace avec le contenu de l'espace mémoire associé à `a`. Le programme finit son exécution en affichant la valeur 1.

A.3 Les commentaires

Toute chaîne de caractères précédée par # dans une ligne est ignorée. Il est donc possible de commenter son code à l'aide de ce mécanisme.

Par exemple, le programme

```
i = 5 # Création d'un variable i
i = i / 2 # Mis a jour de la variable,
          # on voulait en fait une valeur 2 fois plus petite

print( i ) #Affichage de i
```

est équivalent au programme suivant :

```
i = 5
i = i / 2
print( i )
```

A.4 Les entiers, les réels, les booléens et les opérations arithmétiques

Les booléens sont représentés pas les mots clé `True` (pour Vrai) et `False` (pour Faux). Les opérations possibles sont

```
not( BOOLEEN )
BOOLEEN and BOOLEEN
BOOLEEN or BOOLEEN
BOOLEEN == BOOLEEN
```

qui sont respectivement le *non*, le *et*, le *ou* et l'égalité (ou l'équivalence) booléene mathématique.

Nous rappelons que le *non*, le *et*, le *ou* et l'équivalence mathématiques sont définis par :

	True	False
not	False	True

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

⇔ (==)	True	False
True	True	False
False	False	True

Par exemple,

```
bool = False
print( True and bool )
```

affiche `False`

En python, les nombres écrits sans virgule sont des entiers.

De même, les nombres écrits avec une virgule sont des réels. La virgule en python est le ".".

Par exemple,

```
a = 4
b = 4.0
```

la mémoire associée à `a` contient l'entier 4 et la mémoire associée à `b` contient le réel 4.0. Les contenus des espace mémoires de `a` et de `b` sont complètement différents.

Voici les différentes opérations qui existent sur les entiers et les réels.

- L'addition : +
- La soustraction : −
- La multiplication : ∗
- La division réelle ou euclidienne : /
- l'égalité : ==

— la non égalité : ! =

Pour toutes ces opérations, le résultat est un entier si tous les termes sont des entiers. Le résultat est un réel si un des deux termes est un réel.

Enfin, pour la division, si les deux termes sont réels, alors le résultat est le quotient de la division euclidienne des 2 termes. Si l'un des deux termes est un réel, alors le résultat est la division dans le réel des deux termes.

Par exemple, le programme

```
print( 1 + 2 )
print( 1.0 + 2 )
print( 3 / 2 )
print( 3.0 / 2 )
```

affiche respectivement 3, 3.0, 1 et 1.5.

A.5 Les fonctions

Une fonction est une portion de code représentant un sous-programme qui effectue une tâche relativement indépendamment du reste du programme.

Une fonction possède une entrée (les arguments, ou les paramètres) qui constitue les données que l'on donne au départ à la fonction. La fonction exécute ensuite un travail sur ces données, puis retourne éventuellement une valeur appelée sortie du programme.

Toutes les variables définies dans une fonction sont crées dans une espace mémoire indépendant du reste du programme qui est propre à la fonction. L'espace mémoire associée à ces variables est libéré à la fin de l'exécution de la fonction. Les fonctions ne peuvent pas faire appelle à des variables situées en dehors de la définition de la fonction. Le nom des variables définies à l'intérieur de la fonction peut être identique au nom d'autres variables du programme sans affecter leurs états.

En python, on définit les fonctions de la manière suivante :

```
def NOM_FONCTION( PARAMETRE1, PARAMETRE2, ... ):
    INSTRUCTIONS_DE_LA_FONCTION
```

C'est l'indentation qui permet de déterminer les blocs `INSTRUCTIONS_DE_LA_FONCTION`. Toutes les lignes qui à la fois

— commencent avec strictement plus d'espace et de stabulation que la ligne `def NOM_FONCTION( PARAMETRE1, PARAMETRE2, ... ):`

— sont situées juste après `def NOM_FONCTION( PARAMETRE1, PARAMETRE2, ... )`

sont dans `INSTRUCTIONS_DE_LA_FONCTION`.

Pour exécuter le code d'une fonction, il suffit d'écrire dans le programme :

```
NOM_FONCTION( VALEUR1, VALEUR2, ... )
```

Par exemple, le programme;

```
1  def afficher_somme( val1, val2 ):
2    resultat = val1 + val2
3    print( resultat )
4
5  afficher_somme( 3,5 )
6  afficher_somme( 2,1 )
```

afficher 8 puis 3.

Voici le détail de l'exécution de ce programme :

— ligne 1 - 3 : Le programme définit la fonction `afficher_somme`.

— ligne 5 : le programme exécuté `afficher_somme(3,5)` :

    - ligne 1 : Le programme créé un nouvel espace mémoire pour la fonction `afficher_somme`, il alloue dans cet espace mémoire deux petits espaces mémoires et les nomme `val1` et `val2`. Il rempli ensuite le contenu de l'espace mémoire associé à `val1` par le premier entier donnée en paramètre : 3. Puis, il remplit le contenu de l'espace mémoire associé à `val2` par le second entier donné en paramètre : 5.
    - ligne 2 : Le programme alloue dans son espace mémoire personnel un petit espace mémoire, le nomme `resultat`. Il évalue ensuite `val1 + val2`, le contenu de `val1` dans la mémoire de `afficher_somme` vaut 3, celui de `val2` vaut 5, l'évaluation de `val1 + val2` vaut donc 8. Le programme rempli enfin la mémoire de `resultat` par 8.
    - ligne 3 : Le programme affiche le contenu de la mémoire associée à `resultat`, il affiche donc 8.
    - fin de la fonction : la fonction libère son espace mémoire, les données qui y sont contenues sont perdues.
- ligne 6 : le programme exécuté `afficher_somme(1,2)` :
    - ligne 1 : Le programme créé un nouvel espace mémoire pour la fonction `afficher_somme`, il alloue dans cet espace mémoire deux petits espaces mémoires et les nomme `val1` et `val2`. Il rempli ensuite le contenu de l'espace mémoire associé à `val1` par le premier entier donnée en paramètre : 1. Puis, il remplit le contenu de l'espace mémoire associé à `val2` par le second entier donné en paramètre : 2.
    - ligne 2 : Le programme alloue dans son espace mémoire personnel un petit espace mémoire, le nomme `resultat`. Il évalue ensuite `val1 + val2`, le contenu de `val1` dans la mémoire de `afficher_somme` vaut 1, celui de `val2` vaut 2, l'évaluation de `val1 + val2` vaut donc 3. Le programme rempli enfin la mémoire de `resultat` par 3.
    - ligne 3 : Le programme affiche le contenu de la mémoire associée à `resultat`, il affiche donc 3.
    - fin de la fonction : la fonction libère son espace mémoire, les données qui y sont contenues sont perdues.

Il est possible à tout moment, dans la fonction, d'interrompre l'exécution de fonction et de faire retourner une valeur par la fonction. Il suffit d'utiliser l'instruction :

```
return VALEUR_SORTIE
```

Cette instruction arête l'exécution de la fonction et renvoie la valeur `VALEUR_SORTIE`. Ainsi, lors de l'exécution de `NOM_FONCTION( VALEUR1, VALEUR2, ...)`, l'ordinateur substitue `NOM_FONCTION( VALEUR1, VALEUR2, ...)` par l'évaluation de `VALEUR_SORTIE`.

Par exemple, le programme,

```
def f(x):
    a = 4
    return a*x

a = 100
b = f(3)
print( a )
print( b )
```

affiche 100, puis 3.

Voici le détail de l'exécution de ce programme :

- ligne 1 - 3 : Le programme définit la fonction `f`.
- ligne 5 : le programme alloue une petit espace mémoire, le nomme `a` et le rempli avec 100.
- ligne 6 :
    - le programme alloue une petit espace mémoire, le nomme `b`

  - le programme exécute `f(3)` :
    - ligne 1 : Le programme créé un nouvel espace mémoire pour la fonction `f`, il alloue dans cet espace mémoire un petite espace mémoire et le nomme `x`. Il rempli ensuite cet espace avec la valeur donnée en paramètre : 3.
    - ligne 2 : Le programme alloue dans son espace mémoire personnel un petit espace mémoire, le nomme `a` et le rempli par 4.
    - ligne 3 : Le programme évalue a*x, le contenu de `a` dans l'espace mémoire de `f` vaut 4, celui de `x` vaut 3. L'évaluation vaut donc 12. L'instruction `return` interrompt l'exécution de la fonction, renvoie 12 et libère l'espace mémoire de la fonction. Les données qui y étaient stockées sont définitivement perdues.
  - le programme récupère la valeur de retour : 12
  - le programme met 12 dans l'espace mémoire associée à `b`
- ligne 7 : Le programme affiche sur le terminal le contenu de `a` dans la mémoire du programme : il affiche donc 100.
- ligne 8 : Le programme affiche sur le terminal le contenu de `b` dans la mémoire du programme : il affiche donc 12.

Exercice 76
Prévoyez l'exécution du programme suivant :

```
def mystere( a, b ):
    a = a + 2
    b = b + 3
    tmp = a * b
    return tmp
    print( "Calcul de la soustraction" )
    tmp = a - b

a = 3
b = 5
tmp = mystere(a,b)
print( a )
print( b )
print( tmp )
```

A.6 Les saut conditionnels

Les sauts conditionnels permettent à l'ordinateur de choisir entre deux jeux d'instructions, les instructions qu'il va exécuter en fonction d'un test préalable.

```
if TEST :
    LISTE_D_INSTRUCTIONS_1
else :
    LISTE_D_INSTRUCTIONS_2
```

Dans ce code, l'ordinateur évalue l'expression `TEST`. Si cette expression est vraie, alors il exécute les instructions de `LISTE_D_INSTRUCTIONS_1`, sinon il exécute les instructions de `LISTE_D_INSTRUCTION_2`. C'est l'indentation qui permet de déterminer les blocs `LISTE_D_INSTRUCTION_1` et `LISTE_D_INSTRUCTION_2`. Toutes les lignes qui à la fois

- commencent avec strictement plus d'espace et de tabulation que la ligne `if TEST`;

— sont situées juste après `if TEST`

sont dans `LISTE_D_INSTRUCTIONS_1`. Toutes les lignes qui à la fois

— commencent avec strictement plus d'espace et de tabulation que la ligne `else`;

— sont situées juste après `else :`

sont dans `LISTE_D_INSTRUCTIONS_1`. Enfin, les lignes `if` et `else` doivent avoir le même niveau d'indentation.

Par exemple, le programme

```
a = 2
b = 2
if a == b :
    print("a est égal àb")
    print("Comme prévu !")
else :
    print("a est différent de b")
    print( "Ce code ne devrait pas s'exécuter" )
print( "Fin du programme" )
```

affiche

```
a es égal àb
Comme prévu !
Fin du programme
```

Il est possible d'avoir plusieurs sauts conditionnels imbriquées les un dans les autres. Par exemple, le programme

```
if 1 != 2 :
    print( "1 est différent de 2" )
    if 2 == 4 :
        print( "Ne devrait pas s'afficher")
    else:
        print( "2 est différents de 4" )
    print( "Fin du bloc d'instruction" )
else:
    print( "Ne devrait pas s'afficher" )
print( "Fin du programme" )
```

affiche

```
1 est différent de 2
2 est différent de 4
Fin du bloc d'instruction
Fin du programme
```

Lorsque `LISTE_D_INSTRUCTIONS_2` ne contient pas d'instruction, on écrit l'instruction `pass` à la place.

On obtient ainsi :

```
if TEST :
    LISTE_D_INSTRUCTIONS_1
else :
    pass
```

On peut aussi ne pas écrire la partie avec le `else`. Ainsi, le précédent code est équivalent à

```
if TEST :
    LISTE_D_INSTRUCTIONS_1
```

A.7 Les boucles

En programmation, il est utile de pouvoir écrire des programmes qui répètent un certain nombre de fois le même code.

Dans les langages de programmation impératif, ce sont les boucles "for" et les boucles "while" qui permettent de faire exécuter plusieurs fois le même code.

La syntaxe d'une boucle "for" est la suivante :

```
for VARIABLE in range( NOMBRE_DE_REPETITION ):
    LISTE_D_INSTRUCTIONS
```

Il faut savoir qu'en python, `range( NOMBRE_DE_REPETITION )` code la liste `[0,1,2,3,... ,NOMBRE_DE_REPETITION-1]`. Dans les lignes de codes précédentes, l'ordinateur commence par allouer un petit espace mémoire ayant pour nom `VARIABLE`. Ensuite, pour chaque élément e de cette liste, dans l'ordre de la liste, le programme réalise les tâches suivantes :

— il affecte à la variable `VARIABLE` l'élément e;

— il exécute les instructions de `LISTE_D_INSTRUCTIONS`.

A la fin, le programme aura exécuté `NOMBRE_DE_REPETITION` fois les instructions de `LISTE_D_INSTRUCTION`.

Par exemple, le programme suivant compte de 0 à 999 :

```
for compteur in range(1000):
    print compteur
```

La syntaxe d'une boucle while est la suivante :

```
while CONDITION :
    LISTE_D_INSTRUCTIONS
```

Lorsque l'interpréteur python rencontre ces instructions, il commence par évaluer l'expression CONDITION. Si le résultat est Vrai, alors il exécute `LISTE_D_INSTRUCTIONS` puis recommence tout au début, à l'évaluation de l'expression `CONDITION`. Si le résultat est Faux, le programme n'exécute pas `LISTE_D_INSTRUCTIONS` et l'exécution du `while` se termine.

Par exemple, le programme suivant compte de 0 à 999 :

```
compteur = 0
while compteur < 1000 :
    print compteur
  compteur = compteur + 1
```

A.8 Les listes et les tableaux

Une liste d'élément est une suite ordonnée d'éléments. En python, il est possible de coder une liste contenant `N` éléments à l'aide de l'instruction :

```
[ ELEMENT_1, ELEMENT_2, ..., ELEMENT_N ]
```

Lors de l'exécution de cette instruction, l'interpréteur python alloue `N` petit espace mémoire successif dans lequel il place les éléments, `ELEMENT1`, ..., `ELEMENT2`. Puis, l'interpréteur python, substitue `[ELEMENT1, ..., ELEMENTN]` par une référence à la liste crée. Une référence la liste est l'adresse mémoire où se trouve la liste. Une adresse est une donnée de la taille d'un petit espace mémoire. Elle peut donc être stockée dans un petit espace mémoire associée à une variable.

Ainsi, si l'on écrit le programme

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

alors python alloue 3 petits espaces mémoires consécutifs dans la mémoire. Ensuite il substitue `[1,3,2]` par l'adresse mémoire ou se trouve la liste crée. L'interpréteur alloue alors un petit espace mémoire et l'appelle `l`. Enfin, il enregistre dans le petit espace mémoire l'adresse où se trouve la liste précédemment crée.

En python, l'espace mémoire allouée pour les listes ne dépend pas du contexte des fonctions. Ainsi, même si la création de la liste est faite à l'intérieur d'une fonction, la mémoire crée par une liste ne dépend pas de la fonction. Elle n'est donc pas détruite quand la fonction se termine.

La mémoire est détruite automatiquement si elle n'est plus utilisée, c'est à dire si il n'existe plus de référence vers la liste en question.

Par exemple,

```
def creer_tableau( a, b, c ):
    res = [ a,b,c ]
    return res
a = 10
v = creer_tableau( 1,5,6 )
```

Il est possible d'accéder au contenu d'un élément de la liste à l'aide de la commande :

```
LISTE[K]
```

où `LISTE` est le nom de la variable contenant un référence vers une liste et `K` un entier.

Ici, `LISTE[K]` est l'espace mémoire numéro K de la liste qui est située à l'adresse écrite dans le petit espace mémoire associé à `LISTE`. En python les espace mémoire sont numérotés de `0` à la taille de la liste moins 1.

Autrement dit, `LISTE[K]` veut dire :
Dans le petit espace mémoire associé à `LISTE`, vous trouverez une adresse. À cette adresse se trouve une succession de petit espace mémoire. `LISTE[K]` est l'espace mémoire numéro `K`.

Par exemple,

```
l = [-1,20,3]
print( l[0] )
print( l[2] )
```

affiche -1 puis 3 sur le terminal.

Il est possible d'affecter des valeurs à `LISTE[k]` avec l'opérateur d'affectation = en écrivant :

```
LISTE[K] = VALEUR
```

Par exemple,

```
l = [1,3,10]
print( l )
l[1] = 50
print( l )
```

affichera successivement `[1,3,10]` puis `[1,50,10]`

En python il est possible de connaître la taille d'une liste en utilisant la commande;

```
len( LISTE )
```

où `LISTE` est une variable contenant une référence vers une liste.

Par exemple,

```
l = [1,5,2]
print( l )
```

affichera 3 à l'exécution.

Il est possible de concaténer deux listes à l'aide de l'addition.

Ainsi, la variable b du programme suivant

```
B = [1,2,3] + [4,5,6]
```

contiendra une référence vers la nouvelle liste `[1,2,3,4,5,6]`.

Il est possible d'ajouter une nouvelle valeur à la fin de la liste en écrivant :

```
LISTE.append( VALEUR )
```

où `LISTE` est une variable contenant une référence vers une liste et `VALEUR` la valeur que l'on souhaite ajouter à la fin de la liste.

Il est possible de créer une liste à n éléments à l'aide de la commande :

```
[ EXPRESSION(i) for i in range(n)]
```

où `EXPRESSION(i)` est une fonction qui qui dépend de `i`. La liste ainsi obtenue est une liste à n éléments dont l'élément numéro i est l'évaluation de l'expression `EXPRESSION(i)`.

Par exemple,

```
n = 5
l = [ 0 for i in rang(5)]
print(l)
```

affiche `[0,0,0,0,0]`

De même,

```
n = 5
l = [ 2*i for i in range(5)]
print(l)
```

affiche $[0,2,4,6,8]$.

A.9 Les tuples

Les tuples sont identiques aux listes sauf que

— l'on utilise des parenthèses au lieu des crochets;

— les tuples sont immuables, c'est à dire qu'ils ne sont pas modifiables.

Par exemple,

```
b = (1,4,3)
print( b )
print( b[1] )
```

affiche `(1,4,3)` puis 4.

A.10 Le texte et les chaînes de caractères

En python, il est possible de travailler avec des chaînes de caractères. Pour utiliser une chaîne de caractères, il suffit d'écrire :

```
"LA CHAINE DE CARACTERES"
```

Si l'on veut garder un référence vers une chaîne de caractère, il suffit d'écrire :

```
l = "LA CHAINE DE CARACTERE"
```

Dans ce cas, python alloue un petit espace mémoire pour la variable `l`. Ensuite il alloue une espace mémoire plus grand pour enregistré la chaîne de caractères `"LA CHAINE DE CARACTERES"`, enfin il enregistre dans l'espace mémoire de `l` une référence vers l'espace mémoire de la chaîne de caractères.

EXEMPLE

En python les chaînes de caractères sont immuables et uniques! Le caractère immuable veut dire que l'on ne peut pas modifier une chaîne de caractères une fois qu'elle a été crée. Le caractère unique veux dire que si deux chaînes de caractères sont identiques, alors elles auront le même espace mémoire. Par exemple, à la fin du programme,

```
c1 = "Bonjour"
c2 = "Bonjour"
```

`c1` et `c2` contiendront la même référence vers un espace mémoire qui contient la chaîne de caractères "Bonjour".

À contrario, les listes et les tuples ne sont pas uniques. Ainsi le programme suivant

```
c1 = [1,2,3]
c2 = [1,2,3]
```

créera deux variable `c1` et `c2` contenant des références différentes vers des espaces mémoires différents ayant un contenu identique.

B Quelques calculs pour la complexité

B.1 Quelques symboles classiques

On définit les symboles $\mathbb{N}, \mathbb{Z}, \mathbb{R}, \mathbb{R}_+, \mathbb{R}_-, \mathbb{C}$ par :

- $\mathbb{N} = \{0, 1, 2, \ldots\}$ est l'ensemble des entiers positifs;
- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ est l'ensemble des entiers relatifs;
- $\mathbb{R}$ est l'ensemble des réels;
- $\mathbb{R}_+$ est l'ensemble des réels positifs;
- $\mathbb{R}_-$ est l'ensemble des réels négatifs;
- $\mathbb{C}$ est l'ensemble des nombres complexes.

Soit A un sous-ensemble de $\mathbb{C}$, alors A^* est l'ensemble A privé de l'élément neutre 0. Par exemple, $\mathbb{R}_+^*$ est l'ensemble des réels strictement positifs.

B.2 Les sommes et les produits

Soit m et n deux entiers et f une fonction de $\mathbb{N}$ dans $\mathbb{R}$. On note par $\sum_{i=m}^{n} f(i)$ la somme définie par :

$$\boxed{\sum_{i=m}^{n} f(i) = f(m) + f(m+1) + f(m+2) + \cdots + f(n-1) + f(n).}$$

La somme $\sum_{i=m}^{n} f(i)$ vaut 0 si $n < m$.

Par exemple,

$$\sum_{i=3}^{6} i^2 + 2 \times i \quad = \quad (3^2 + 2 \times 3) \quad + \quad (4^2 + 2 \times 4) \quad + \quad (5^2 + 2 \times 5) \quad + \quad (6^2 + 2 \times 6).$$

On note par $\prod_{i=m}^{n} f(i)$ le produit défini par :

$$\boxed{\prod_{i=m}^{n} f(i) = f(m) \times f(m+1) \times f(m+2) \times \cdots \times f(n-1) \times f(n).}$$

Le produit $\prod_{i=m}^{n} f(i)$ vaut 1 si $n < m$.

Par exemple,

$$\prod_{i=3}^{6} i^2 + 2 \times i \quad = \quad (3^2 + 2 \times 3) \quad \times \quad (4^2 + 2 \times 4) \quad \times \quad (5^2 + 2 \times 5) \quad \times \quad (6^2 + 2 \times 6).$$

Pour tout entier $n \geq 0$, on défini la factorielle de n par :

$$\boxed{n! = \prod_{i=1}^{n} i = 1 \times 2 \times \ldots \times n}$$

Par exemple $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$ et $0! = 1$.

B.3 Les binomiaux

Soit m et n deux entiers tels que $m \leq n$. Les binomiaux C_n^m sont des entiers définis par :

$$\boxed{C_n^m = \frac{n!}{m!(n-m)!}.}$$

Les binomiaux vérifient la propriété suivante :

$$\boxed{C_n^m = C_{n-1}^{m-1} + C_{n-1}^m} \tag{10}$$

En effet,

$$C_{n-1}^{m-1} + C_{n-1}^m = \frac{(n-1)!}{(m-1)!(n-m)!} + \frac{(n-1)!}{m!(n-1-m)!} = \frac{(n-1)!m + (n-1)!(n-m)}{m!(n-m)!} = \frac{n!}{m!(n-m)!} = C_n^m.$$

On aime bien généralement représenter cette propriété sous la forme d'un triangle appelé triangle de Pascal qui est présenté à la figure 80.

$$\begin{array}{ccccccccc} & & & & C_0^0 & & & & \\ & & & C_1^0 & & C_1^1 & & & \\ & & C_2^0 & + & C_2^1 & & C_2^2 & & \\ & C_3^0 & & C_3^1 & & C_3^2 & & C_3^3 & \\ C_4^0 & & C_4^1 & & C_4^2 & & C_4^3 & & C_4^4 \end{array} \qquad \begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & + & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \end{array}$$

FIGURE 80 – Le triangle de Pascal

Les binomiaux jouent un rôle important en mathématiques car il apparaissent dans le développement de

$$\boxed{(a+b)^n = \sum_{k=0}^{n} C_n^k a^k b^{n-k}} \tag{11}$$

Cette formule se démontre par récurrence sur n de la façon suivante :

— Si $n = 0$, alors, $(a+b)^0 = 1$ et $C_0^0 a^0 b^0 = 1$, donc la formule est vrai;

— Soit $M > 0$ un entier. Supposons que la formule soit vraie pour $n \leq M$, est-elle vrai pour $n = M+1$? Développons $(a+b)^{M+1}$:

$$\begin{aligned}
(a+b)^{M+1} &= (a+b).(a+b)^M = (a+b)\sum_{k=0}^{M} C_M^k a^k b^{M-k} \\
&= \sum_{k=0}^{M} C_M^k a^{k+1} b^{M-k} + \sum_{k=0}^{M} C_M^k a^k b^{M+1-k} \\
&= a^{M+1} + \sum_{k=1}^{M}\left(C_M^{k-1} + C_M^k\right).a^k b^{M+1-k} + b^{M+1}
\end{aligned}$$

D'après l'équation 10, on sait que $C_{M+1}^k = C_M^{k-1} + C_M^k$, on a alors :

$$\begin{aligned}
(a+b)^{M+1} &= a^{M+1} + \sum_{k=1}^{M} C_{M+1}^k.a^k b^{M+1-k} + b^{m+1} \\
&= \sum_{k=0}^{M+1} C_{M+1}^k a^k b^{M+1-k}.
\end{aligned}$$

Ce qui finit de démontrer par récurrence la formule 11.

B.4 Calcul de sommes classiques

Proposition 1. *Soit n un entier et r un réel différent de 1, alors*

$$\boxed{\sum_{k=0}^{n} r^k = \frac{1-r^{n+1}}{1-r}}$$

Démonstration.

$$\begin{aligned}
(1-r).\textstyle\sum_{k=0}^{n} r^k &= (1 + r + r^2 + \cdots + r^n) - (r + r^2 + \cdots + r^n + r^{n+1}) \\
&= 1 - r^{n+1}
\end{aligned}$$

□

Proposition 2. *Soit n un entier, alors*

$$\boxed{\sum_{k=0}^{n} k = \frac{n(n+1)}{2} \qquad \sum_{k=0}^{n} k^2 = \frac{(2n+1)(n+1)n}{6} \qquad \sum_{k=0}^{n} k^3 = \frac{n^2(n+1)}{4}}$$

Démonstration. La formule $S_n = \sum_{k=0}^{n} k = \frac{n(n+1)}{2}$ est célèbre. Elle a été démontré par Carl Friedrich Gauss de la façon suivante :

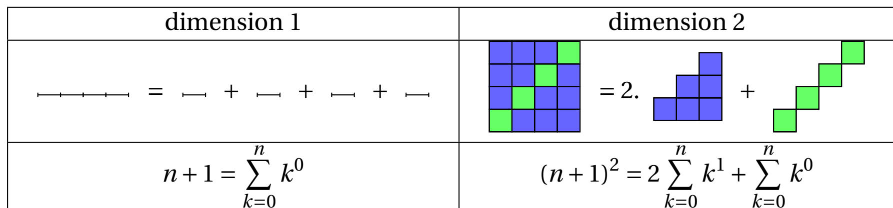
$$\begin{array}{ccccccccccc}
S_n & = & 1 & + & 2 & + & \dots & + & n-1 & + & n \\
S_n & = & n & + & n-1 & + & \dots & + & 2 & + & 1 \\
\hline
2.S_n & = & n+1 & + & n+1 & + & \dots & + & n+1 & + & n+1
\end{array}$$

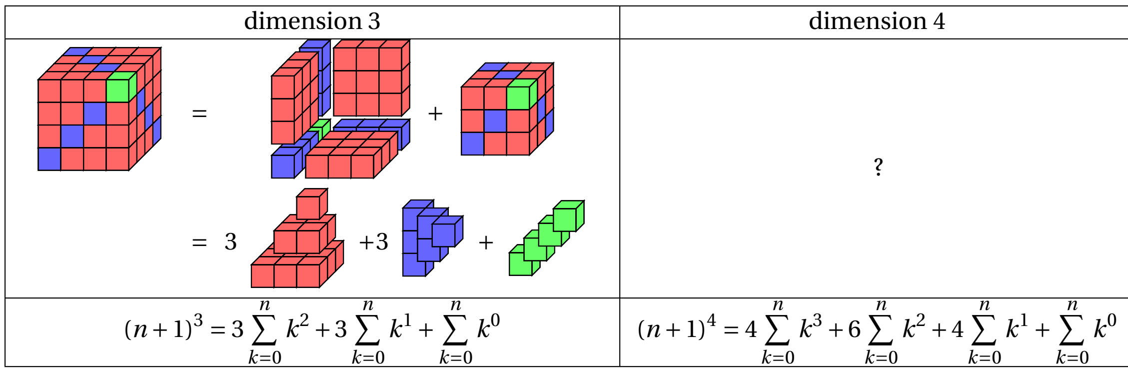
$$2.S_n = n(n+1)$$

□

Cependant, cette preuve ne se généralise pas pour le calcul des sommes $\sum_k k^d$ où d est un entier quelconque.

En fait, cette identité remarquable est un cas particulier d'une identité géométrique qui est la suivante :

dimension 1	dimension 2
	
$n+1=\sum_{k=0}^{n} k^0$	$(n+1)^2 = 2\sum_{k=0}^{n} k^1 + \sum_{k=0}^{n} k^0$

dimension 3	dimension 4
	?
$(n+1)^3 = 3\sum_{k=0}^{n} k^2 + 3\sum_{k=0}^{n} k^1 + \sum_{k=0}^{n} k^0$	$(n+1)^4 = 4\sum_{k=0}^{n} k^3 + 6\sum_{k=0}^{n} k^2 + 4\sum_{k=0}^{n} k^1 + \sum_{k=0}^{n} k^0$

En regardant les formules des tableaux précédents, on reconnaît les premières lignes du triangle de Pascal, présenté à la figure 80. On peut donc facilement en déduire une formule conjecturale pour la dimension 4, (voir tableau précédent), et on peut traduire ces observations mathématiquement par, pour tout entier n et toute dimension d,

$$(n+1)^d = \sum_{i=0}^{d-1} C_d^i \sum_{k=0}^{n} k^i. \tag{12}$$

Il ne reste plus qu'à démontrer cette dernière identité en procédant de la façon suivante : On sait que $(k+1)^d = \sum_{i=0}^{d} C_d^i k^i$. Ainsi, si l'on somme cette dernière identité pour k allant de 0 à n, on obtient que :

$$\sum_{k=0}^{n} (k+1)^d = \sum_{k=0}^{n} \sum_{i=0}^{d} C_d^i k^i = \sum_{i=0}^{d} \sum_{k=0}^{n} C_d^i k^i = \sum_{i=0}^{d} C_d^i \sum_{k=0}^{n} k^i.$$

Si l'on extrait le terme $i = d$ de la somme du dernier terme, on obtient :

$$\sum_{k=0}^{n} (k+1)^d = \sum_{k=0}^{n} k^d + \sum_{i=0}^{d-1} C_d^i \sum_{k=0}^{n} k^i$$

qui se simplifie de part et d'autre de l'égalité pour donner :

$$(n+1)^d = \sum_{i=0}^{d-1} C_d^i \sum_{k=0}^{n} k^i.$$

On peut maintenant utiliser l'équation 12 pour déterminer récursivement les sommes cherchées :

$$(n+1) = \sum_{k=0}^{n} k^0 \implies \sum_{k=0}^{n} k^0 = n+1$$

$$(n+1)^2 = 2.\sum_{k=0}^{n} k + \sum_{k=0}^{n} k^0 \implies \sum_{k=0}^{n} k = \frac{(n+1)^2-(n+1)}{2}$$

$$(n+1)^3 = 3.\sum_{k=0}^{n} k^2 + 3.\sum_{k=0}^{n} k + \sum_{k=0}^{n} k^0 \implies \sum_{k=0}^{n} k^2 = \frac{1}{3}\left((n+1)^3 - 3.\frac{n(n+1)}{2} - (n+1)\right)$$

$$(n+1)^4 = 4.\sum_{k=0}^{n} k^3 + 6.\sum_{k=0}^{n} k^2 + 4.\sum_{k=0}^{n} k + \sum_{k=0}^{n} k^0 \implies \sum_{k=0}^{n} k^3 = \frac{(n+1)^4 - 6.\frac{(2n+1)(n+1)n}{6} - 4.\frac{n(n+1)}{2} - (n+1)}{4}.$$

Trouver une formule close (difficile, pour les curieux) :

Commençons par poser la matrice $M = \left(m_{i,j}\right)$ définie par :

$$M = \begin{cases} C_i^{j-1} & \text{si } i \geq j \\ 0 & \text{sinon.} \end{cases}$$

Si on note par $\overrightarrow{s}$ et $\overrightarrow{n}$ les vecteurs :

$$\overrightarrow{s} = \left(\sum_{k=0}^{n} k^i\right)_{0\leq i\leq d} \qquad \overrightarrow{n} = \left((n+1)^i\right)_{1\leq i\leq d+1}$$

alors le problème consistant à chercher une formule pour $\sum_{l=0}^{n} l^i$ se réécrit par :

$$M.\overrightarrow{s} = \overrightarrow{n}$$

Par exemple, pour $d = 4$, on a :

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \\ 1 & 3 & 3 & 0 & 0 \\ 1 & 4 & 6 & 4 & 0 \\ 1 & 5 & 10 & 10 & 5 \end{pmatrix} \qquad \overrightarrow{s} = \begin{pmatrix} \sum_{k=0}^{n} k^0 \\ \sum_{k=0}^{n} k^1 \\ \sum_{k=0}^{n} k^2 \\ \sum_{k=0}^{n} k^3 \\ \sum_{k=0}^{n} k^4 \end{pmatrix} \qquad \overrightarrow{n} = \begin{pmatrix} (n+1) \\ (n+1)^2 \\ (n+1)^3 \\ (n+1)^4 \\ (n+1)^5 \end{pmatrix}$$

et l'on a donc la relation suivante :

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \\ 1 & 3 & 3 & 0 & 0 \\ 1 & 4 & 6 & 4 & 0 \\ 1 & 5 & 10 & 10 & 5 \end{pmatrix} . \begin{pmatrix} \sum_{k=0}^{n} k^0 \\ \sum_{k=0}^{n} k^1 \\ \sum_{k=0}^{n} k^2 \\ \sum_{k=0}^{n} k^3 \\ \sum_{k=0}^{n} k^4 \end{pmatrix} = \begin{pmatrix} (n+1) \\ (n+1)^2 \\ (n+1)^3 \\ (n+1)^4 \\ (n+1)^5 \end{pmatrix}.$$

La matrice M étant triangulaire, elle s'inverse donc.

Soit $T = \left(t_{i,j}\right)$ une matrice triangulaire dont la diagonale est constituée uniquement de 1 ($t_{i,i} = 1$). L'inverse $S = \left(s_{i,j}\right)$ d'une telle matrice se calcule facilement en utilisant la formule de la comatrice et donne :

$$s_{i,j} = \begin{cases} \displaystyle\sum_{k=1}^{i-j} \sum_{\substack{u_1+u_2+\cdots+u_k=i-j \\ 1\leq u_1,\ldots,u_k\leq i-j}} (-1)^k \prod_{i=1}^{k} t_{j+\sum_{l=1}^{i} u_l,\ j+\sum_{l=1}^{i-1} u_l} & \text{si } i > j \\ 1 & \text{si } i = j \\ 0 & \text{sinon} \end{cases}$$

Si on applique cette dernière formule à notre matrice M, on obtient alors que l'inverse M^{-1} de M est donné par :

$$\left(M^{-1}\right)_{i,j} = \frac{1}{C_i^1} \begin{cases} \displaystyle\sum_{k=1}^{i-j} \sum_{\substack{u_1+u_2+\cdots+u_k=i-j \\ 1\leq u_1,\ldots,u_k\leq i-j}} (-1)^k \frac{C_{j+u_1}^{j-1}}{C_j^1} \frac{C_{j+u_1+u_2}^{j-1+u_1}}{C_{j+u_1}^1} \cdots \frac{C_{j+u_1+\cdots+u_k}^{j-1+u_1+\cdots+u_{k-1}}}{C_{j+u_1+\cdots+u_{k-1}}^1} & \text{si } i > j \\ 1 & \text{si } i = j \\ 0 & \text{sinon} \end{cases}$$

En remarquant que $j + u_1 + \dot{+} u_k = i$ et que

$$\frac{C_{j+u_1}^{j-1}}{C_j^1} \frac{C_{j+u_1+u_2}^{j-1+u_1}}{C_{j+u_1}^1} \cdots \frac{C_{j+u_1+\cdots+u_k}^{j-1+u_1+\cdots+u_{k-1}}}{C_{j+u_1+\cdots+u_{k-1}}^1} = \frac{i!}{j!} \frac{1}{(u_1+1)!.(u_2+1)!.\ldots.(u_k+1)!}$$

On obtient finalement la formule suivante :

$$\left(M^{-1}\right)_{i,j}=\begin{cases}\dfrac{(i-1)!}{j!}\displaystyle\sum_{k=1}^{i-j}(-1)^k\sum_{\substack{u_1+u_2+\cdots+u_k=i-j\\1\le u_1,\ldots,u_k\le i-j}}\frac{1}{(u_1+1)!(u_2+1)!\ldots(u_k+1)!}\\ 1/i & \text{si } i=j\\ 0 & \text{sinon}\end{cases}$$

Il ne reste plus qu'à développer la dernière ligne de $\vec{s}=M^{-1}.\vec{n}$ pour obtenir une formule close pour $\sum k^d$:

$$\sum_{k=0}^{n}k^d=\frac{1}{d+1}(n+1)^{d+1}+\sum_{j=1}^{d}\left(\frac{d!}{j!}\sum_{k=1}^{d+1-j}(-1)^k\sum_{\substack{u_1+u_2+\cdots+u_k=d+1-j\\1\le u_1,\ldots,u_k\le d+1-j}}\frac{1}{(u_1+1)!(u_2+1)!\ldots(u_k+1)!}\right)(n+1)^j$$

Tout faire en 2 lignes avec l'exponentielle :

Proposition 3. *Soit n et d deux entiers positifs, alors,*

$$\sum_{k=0}^{n}k^d=\left(\frac{\partial^d}{\partial x^d}\frac{e^{(n+1)x}-1}{e^x-1}\right)(0).$$

où $\frac{\partial^d}{\partial x^d}f$ est la dérivée $d^{\text{ième}}$ de f.

Démonstration. On reconnaît dans $\frac{e^{(n+1)x}-1}{e^x-1}$ le calcul d'une somme géométrique de raison e^x. Ainsi,

$$\frac{\partial^d}{\partial x^d}\frac{e^{(n+1)x}-1}{e^x-1}=\frac{\partial^d}{\partial x^d}\sum_{k=0}^{n}e^{k.x}=\sum_{k=0}^{n}k^d e^{k.x}.$$

Si l'on évalue l'expression précédente en 0, on obtient,

$$\frac{\partial^d}{\partial x^d}\frac{e^{(n+1)x}-1}{e^x-1}(0)=\sum_{k=0}^{n}k^d.$$

□

Proposition 4. *La somme $\sum_{k=0}^{n}k^d$ est un polynôme de degrés $d+1$ en n et vaut :*

$$\sum_{k=0}^{n}k^d=\frac{1}{d+1}\sum_{j=1}^{d+1}\left(C_{d+1}^{j}.\left(\frac{\partial^{d+1-j}}{\partial x^{d+1-j}}\frac{1}{1-g(x)}\right)(0)\right).(n+1)^j$$

où $g(x)=(1+x-e^x).x^{-1}$.

Démonstration. Soit $f\in\mathscr{C}^{d+1}$, alors

$$\frac{\partial^{d+1}}{\partial x^{d+1}}\left(xf\right)=\sum_{j=0}^{d+1}C_{d+1}^{j}\frac{\partial^j}{\partial x^j}x\frac{\partial^{d+1-j}}{\partial x^{d+1-j}}f=x\frac{\partial^{d+1}}{\partial x^{d+1}}f+(d+1)\frac{\partial^d}{\partial x^d}f.$$

Ainsi,

$$\frac{\partial^{d+1}}{\partial x^{d+1}}\left(xf\right)(0)=(d+1)\frac{\partial^d}{\partial x^d}f(0).$$

Comme

$$\frac{e^{(n+1)x}-1}{e^x-1}=\sum_{k=0}^{n}e^{k.x}$$

est $\mathscr{C}^{d+1}$ sur $\mathbb{R}$, on peut utiliser la formule précédente en posant $f = \frac{e^{(n+1)x}-1}{e^x-1}$ pour obtenir :

$$\frac{\partial^d}{\partial x^d} \frac{e^{(n+1)x}-1}{e^x-1}(0) = \frac{1}{d+1} \frac{\partial^{d+1}}{\partial x^{d+1}} \frac{x\left(e^{(n+1)x}-1\right)}{e^x-1}(0) = \frac{1}{d+1} \frac{\partial^{d+1}}{\partial x^{d+1}} \frac{e^{(n+1)x}-1}{1-g(x)}(0)$$

Si l'on développe cette dernière expression, la somme $\sum_{k=0}^{n} k^d$ devient

$$\frac{1}{d+1} \sum_{j=0}^{d+1} C_{d+1}^{j} \frac{\partial^j}{\partial x^j} \left(e^{(n+1)x}-1\right) \frac{\partial^{d+1-j}}{\partial x^{d+1-j}} \frac{1}{1-g(x)}(0)$$

L'évaluation en 0 de $\frac{\partial^j}{\partial x^j}\left(e^{(n+1)x}-1\right)$ vaut 0 si $j=0$ et $(n+1)^j$ sinon. Ainsi, la somme $\sum_{k=0}^{n} k^d$ est égale à

$$\frac{1}{d+1} \sum_{j=1}^{d+1} C_{d+1}^{j} \frac{\partial^{d+1-j}}{\partial x^{d+1-j}} \frac{1}{1-g(x)}(0)(n+1)^j.$$

□