Numérique et Science Informatique Terminale

Alain Busser

2 juillet 2025

Chapitre 1

Les Graphes

1.1 Dictionnaires

1.1.1 Clés et valeurs

Dans un dictionnaire `Python`, chaque clé est associée à une valeur unique. Dans le dictionnaire `dico`, on obtient la valeur associée à la clé `clef` par l'expression `dico[clef]`.

1.1.2 Valeurs

L'expression `dico.values()` permet de lister les valeurs du dictionnaire.

1.1.3 Clés

L'expression `dico.keys()` permet de lister les clés du dictionnaire.
Le parcours d'un dictionnaire se fait par les clés :

```
for k in dico:
    print(k,dico[k])
```

1.2 Graphes orientés

1.2.1 Définition

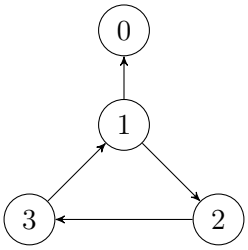
Un *graphe orienté* est un dictionnaire dont les valeurs sont des listes de clés.

```
go = {
    0: [],
    1: [0,2],
    2: [3],
    3: [1]
}
```

Les clés du dictionnaire s'appellent les *sommets* du graphe. Si b est dans la liste d'adjacence de a on dit que a et b sont *adjacents* et que $a \longrightarrow b$ est un *arc* (ou une *arête*) du graphe.

1.2.2 Représentation

Cette représentation est dite *par listes d'adjacences*.
On peut aussi représenter un graphe par un dessin :



1.2.3 Matrice d'adjacence

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

1.3 Graphes

1.3.1 Définition

Si chaque fois qu'il y a une arête allant de a vers b il y en a aussi une allant de b vers a alors le graphe est dit *non orienté* (ou *graphe*).

```
g = {
    0: [1],
    1: [0,2,3],
    2: [1,3],
    3: [1,2]
}
```

La matrice d'un graphe non orienté, est symétrique (et réciproquement) :

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

1.3.2 Parcours

On parcourt un graphe à partir d'un sommet. Pour un *parcours en largeur* on traite toute la liste d'adjacence d'un sommet avant de passer au sommet suivant. Pour un *parcours en profondeur* dès qu'on découvre un nouveau sommet on regarde si dans sa liste d'adjacence il n'y a pas de sommet non encore exploré.

1.3.3 Chemins

Un *chemin* est une liste de sommets tels que chacun est adjacent au suivant (on suppose qu'il n'y a pas deux fois la même arête). Un *cycle* est un chemin dont le départ et l'arrivée coïncident.
Un graphe sans cycle s'appelle un *arbre*.

Chapitre 2

Routage

2.1 Notation CIDR

2.1.1 Classless Inter-Domain Routing

Une adresse IPv4 est formée de 4 octets (32 bits) écrits en décimal, séparés par des points.
Le masque de réseau est noté de la même manière, par exemple pour n=24, `255.255.255.0`
La machine d'adresse IP `243.1.2.5/24` utilise ce masque de réseau.

2.1.2 Adresse de réseau

Le masquage produit l'adresse IP `243.1.2.0` qui est l'*adresse réseau* (commune à toutes les machines de ce réseau).

2.1.3 Adresse de diffusion

Un fichier envoyé à l'adresse `243.1.2.5` sera reçu par la machine. Mais un fichier envoyé à l'adresse `243.1.2.255` aussi : c'est l'*adresse de diffusion* (*broadcast*) du réseau.

2.1.4 Table de routage

Une *table de routage* est un tableau à 5 colonnes, stocké dans le routeur, les descripteurs étant :

- l'adresse IP de destination
- le masque de réseau correspondant
- l'adresse IP de la passerelle
- l'adresse IP de l'interface
- la métrique correspondante.

On lit une table de routage avec

```
ip -4 route
```

2.2 Protocole RIP

2.2.1 Routing Information Protocol

Dans RIP, la métrique est le nombre de sauts (*hops*) entre les routeurs : 1 pour les voisins du routeur, etc.
`traceroute` utilise ce protocole.

2.2.2 Exemple

table de routage 0

destination	interface	métrique
1	1	1
2	1	2
3	1	2

table de routage 1

destination	interface	métrique
0	0	1
2	2	1
3	3	1

table de routage 2

destination	interface	métrique
0	1	2
1	1	1
3	3	1

table de routage 3

destination	interface	métrique
0	1	2
1	1	1
2	2	1

2.3 Protocole OSPF

2.3.1 Open Shortest Path First

OSPF tient compte du *coût* qui est un entier associé à chaque liaison entre routeurs.
Pour CISCO, le coût est défini comme le quotient de 10^8 bits/seconde, par le débit de la liaison, arrondi à l'entier supérieur. Le plus petit coût est donc 1.

2.3.2 Exemple

Si la liaison 1-3 est de coût 10 (les autres liaisons étant de coût 1), les tables de routage deviennent :
table de routage 0

destination	interface	métrique
1	1	1
2	1	2
3	1	3

table de routage 1

destination	interface	métrique
0	0	1
2	2	1
3	2	2

table de routage 2

destination	interface	métrique
0	1	2
1	1	1
3	1	2

table de routage 3

destination	interface	métrique
0	2	3
1	2	2
2	2	1

Chapitre 3

Arbres

3.1 Définitions

3.1.1 Arbre

Un *arbre* est un graphe sans cycle.
Les sommets sont appelés *nœuds* et les arêtes sont appelées *branches*.
Le nombre de nœuds est la *taille* de l'arbre.

3.1.2 Racine

On choisit un nœud à partir duquel on effectue les parcours dans l'arbre. Il s'appelle la *racine* de l'arbre.

3.1.3 feuilles

Un nœud dont il ne part aucune branche est une *feuille* de l'arbre.

3.2 Exemples

3.2.1 labyrinthes

Un labyrinthe est modélisé par un graphe. Explorer le labyrinthe revient à trouver un *arbre couvrant* (qui a les mêmes sommets). Pour cela on peut effectuer un parcours en largeur ou en profondeur. Il y a plusieurs sortes de parcours en profondeur :

- Si on traite chaque nouveau nœud au moment où on le découvre, le parcours est *préfixe*.
- Si on traite chaque nœud après avoir traité tous ses descendants, le parcours est *suffixe*.

3.2.2 `tree`

La commande `tree` affiche dans le terminal l'arborescence des fichiers.

3.2.3 `pstree`

La commande `pstree` affiche dans le terminal les processus en tenant compte de leur parent.

Chapitre 4

Récursivité

4.1 Définition

Une fonction Python est dite *récursive* si au moins un appel à cette fonction figure dans la définition de la fonction.

```
def recfunc(args):
    ...
    a = ...
    ...
    return expr(...,recfunc(a))
```

4.2 Cas de base

Pour être calculable en temps fini, une fonction récursive doit avoir un *cas de base* :

```
def recfunc(variant1):
    if cas_de_base():
        return valeur_de_base
    else:
        return recfunc(variant2)
```

Pour prouver la terminaison d'une fonction récursive, on utilise un *variant* (comme pour les boucles).

4.3 Exemples

4.3.1 Conversion binaire

```
def binaire(n:int)->str:
    assert n>=0 and type(n)==int
    if n<2:
        return str(n)
    else:
        return binaire(n//2)+str(n%2)
```

4.3.2 Somme d'une liste

```
def somme(L):
    if L==[]:
        return 0
    else:
        return L[0]+somme(L[1:])
```

Chapitre 5

Arbres binaires

5.1 Définition

Un arbre est binaire si chaque nœud possède au maximum deux enfants :

- l'enfant gauche qui est un arbre binaire (éventuellement vide),
- l'enfant droit qui est aussi un arbre binaire (éventuellement vide).

Comme on distingue les sous-arbres gauche et droit, un arbre binaire n'est pas considéré comme un arbre.

5.2 Modélisation Python

5.2.1 Classe arbre binaire

```
class Arbre():
    def __init__(self,nom):
        self.label = nom
        self.G = None
        self.D = None
    def est_une_feuille(self):
        return self.G is None and self.D is None
    def greffe_gauche(self,greffe):
        self.G = greffe
    def greffe_droite(self,greffe):
        self.D = greffe
```

5.2.2 implémentation d'un graphe

```
graphe = Arbre(0)
graphe.greffe_gauche(Arbre(1))
graphe.G.greffe_gauche(Arbre(2))
graphe.G.G.greffe_gauche(Arbre(3))
graphe.greffe_droite(Arbre(1))
graphe.G.greffe_droite(Arbre(2))
graphe.G.G.greffe_droite(Arbre(3))
graphe.G.G.G.greffe_droite(Arbre(1))
```

5.3 Parcours

5.3.1 Parcours en largeur

Un parcours en largeur à partir de la racine classe les nœuds par *niveau*. Le niveau le plus élevé s'appelle la *hauteur* de l'arbre.

5.3.2 Parcours en profondeur

On distingue trois parcours en profondeur :

- le parcours *préfixe*, où le nœud est traité avant ses enfants,
- le parcours *suffixe*, où le nœud est traité après ses enfants,
- le parcours *infixe*, où le nœud est traité après son enfant gauche mais avant son enfant droit.

5.4 Mesures

5.4.1 Taille

La *taille* d'un arbre binaire est le nombre de nœuds qu'il contient (racine comprise).

```
def taille(arbre):
    if arbre is None:
        return 0
    else:
        return 1+taille(arbre.G)+taille(arbre.D)
```

Le nombre de branches d'un arbre binaire est un de moins que sa taille.

5.4.2 Hauteur

La hauteur d'un arbre binaire est calculable récursivement :

```
def hauteur(arbre):
    if arbre is None:
        return -1
    else:
        return 1+max(hauteur(arbre.G),hauteur(arbre.D))
```

5.4.3 Comparaison

La taille n d'un arbre binaire de hauteur h est comprise entre $h+1$ et $2^{h+1}-1$.
La hauteur h est comprise entre $log_2(n-1)$ et $n-1$, où log_2 est définie récursivement par

```
def log2(n):
    if n<2:
        return n
    else:
        return 1+log2(n//2)
```

Chapitre 6

Bases de données

6.1 Modèle relationnel

6.1.1 Relation

Une *relation* (ou *table*) est un tableau dont la première ligne (*headers* ou descripteurs) est formée de chaînes de caractères :

idvaleur	points
as	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1
valet	2
dame	4
roi	5

6.1.2 Schéma

Le schéma d'une table indique les entrées de la table dans l'ordre et avec leur type.
On peut donner le schéma d'une table en SQL :

```
CREATE TABLE valeurs (idvaleur TEXT PRIMARY KEY, points INT);
```

ou graphiquement :

valeurs	
idvaleur	TEXT
points	INT

6.1.3 Enregistrements

Chaque ligne de la table s'appelle un *enregistrement*. C'est un n-uplet nommé comme `'as',1)`.

```
INSERT INTO valeurs VALUES ('as',1);
INSERT INTO valeurs VALUES ('2',1);
```

```
...
INSERT INTO valeurs VALUES ('roi',5);
```

Contraintes d'intégrité

```
INSERT INTO valeurs VALUES (5);
```

donne un message d'erreur : une *contrainte de relation* n'est pas respectée.

```
INSERT INTO valeurs VALUES (2,1);
```

ne respecte pas une *contrainte de domaine*.
Pour modifier un enregistrement, on peut faire

```
DELETE FROM valeurs WHERE idvaleur='roi';
INSERT INTO valeurs VALUES ('roi',5);
```

ou

```
UPDATE valeurs SET points=5 WHERE idvaleur='roi';
```

6.1.4 Champs

Un *champ* (ou colonne) est décrit par son *attribut*.
Les attributs de `valeurs` sont `valeurs.idvaleur` et `valeurs.points`.
Pour créer un tableau d'une seule colonne on fait

```
SELECT idvaleur FROM valeurs;
```

ou

```
SELECT points FROM valeurs;
```

6.2 Clés

6.2.1 Définition

Une *clé* est un n-uplet d'attributs.
`idvaleur` est une clé. `points` est une clé. `idvaleur,points` et `points,idvaleur` aussi.
La clé comprenant tout le schéma est notée `*` en SQL.

6.2.2 clé primaire

Une *clé primaire* est une clé ne prenant pas plusieurs fois la même valeur.
`points` n'est pas une clé primaire parce que l'as et le 2 valent tous deux 1 point.
`idvaleur` est une clé primaire. Dans le schéma, elle est soulignée.

Contrainte d'intégrité

```
INSERT INTO valeurs VALUES ('roi',5);
INSERT INTO valeurs VALUES ('roi',4);
```

ne respecte pas la contrainte d'unicité d'une clé primaire.

6.3 Requêtes d'interrogation

6.3.1 Combien un roi rapporte-t-il de points ?

```
SELECT points FROM valeurs WHERE idvaleur='roi';
```

6.3.2 Nombre de points possibles

```
SELECT DISTINCT points FROM valeurs;
```

6.3.3 Quelles sont les figures ?

```
SELECT idvaleur FROM valeurs WHERE points >= 2;
```

6.3.4 Combien de cartes ?

```
SELECT COUNT(*) FROM valeurs;
```

6.3.5 Trier les valeurs

```
SELECT idvaleur FROM valeurs ORDER BY points;
```

6.3.6 Combien de points en moyenne ?

```
SELECT AVG(points) FROM valeurs;
```

6.4 SGBD

6.4.1 Définition

Un Système de Gestion de Bases de Données (SGBD) est un logiciel permettant d'utiliser une base de données.
Exemple : `sqlite3`.

6.4.2 Propriétés

Un SGBD doit permettre :

- la persistance des données,
- la gestion des accès concurrents,
- l'efficacité de traitement des requêtes,
- la sécurisation des accès.

Chapitre 7

Listes chaînées

7.1 Définition

(Allen Newell 1955, John McCarthy 1958)
Une *liste chaînée* est un arbre binaire dans lequel chaque nœud n'a qu'un enfant droit.
Les nœuds s'appellent des *cellules*.
La racine est le *premier élément* de la liste chaînée.
L'enfant droit d'un nœud est *le nœud suivant*.
La feuille (nœud n'ayant pas d'enfant droit) est *le dernier élément* de la liste chaînée.

7.2 Algorithmes

```
class Liste:
    def __init__(self,label):
        self.label = label
        self.suivant = None
    def est_final(self):
        return self.suivant is None
    def suite(self):
        return self.suivant
    def ajout(self,elt):
        if self.est_final():
            self.suivant = Liste(elt)
        else:
            self.suivant.ajout(elt)
```

7.2.1 Longueur

```
def longueur(liste):
    if liste is None:
        return 0
    else:
        return 1+longueur(liste.suite())
```

7.2.2 Appartenance

```
def contient(liste,elt):
    if liste is None:
        return False
    else:
        return liste.label==elt or contient(liste.suite(),elt)
```

Chapitre 8

Processus

8.1 Ordonnancement

8.1.1 Système d'exploitation

Le système d'exploitation est un logiciel qui gère les ressources matérielles de l'ordinateur (CPU, mémoire, entrées-sorties,...) et sert d'interface aux autres logiciels.

8.1.2 Processus

Un *processus* est un programme en cours d'exécution.
Pour créer un processus, le système d'exploitation place une copie du programme en mémoire (RAM), réserve une place en mémoire pour la pile et pour d'autres ressources (variables, moyen de communiquer avec d'autres processus). Une fois créé, le processus est dans l'état `sleeping`.

8.1.3 états

Un processus peut être dans l'état

- *en veille* (`sleeping`)
- *en cours* (`running`)
- *en attente* (`stopped`) s'il a besoin d'une ressource indisponible
- *terminé*

8.1.4 ordonnancement

Le système d'exploitation met les processus (dans l'état `stopped` ou `sleeping`) dans une file, et leur attribue l'un après l'autre un *quantum* de temps. Lorsqu'un processus est en tête de file,

- il est mis dans l'état `running` durant un quantum,
- s'il n'est pas terminé, il est remis dans la file.

8.2 Interblocage

8.2.1 état `stopped`

Lorsqu'un processus est dans l'état `stopped`, il attend une ressource. Si cette ressource se libère il la réserve pour lui.

8.2.2 deadlock

(Coffman 1971)
Lorsque deux processus P1 et P2 ont tous les deux besoin de deux ressources R1 et R2 et que

- P1 a réservé R1 et attend R2
- P2 a réservé R2 et attend R1

alors il y a *interblocage*.

Chapitre 9

Piles

9.1 Définitions

9.1.1 Pile

(Turing 1950)
Une *pile* est une liste accessible uniquement par la fin (son dernier élément).
Son dernier élément est *le sommet* de la pile.
Le nombre d'éléments est *la hauteur* de la pile.
Ajouter un élément (au sommet) se dit *empiler*. C'est une instruction.
Retirer le dernier élément se dit *dépiler*. C'est une expression.

9.1.2 LIFO

Last In, First Out décrit le fonctionnement d'une pile.

9.2 Classe Pile

```
class Pile:
    def __init__(self):
        self.tab = []
    def est_vide(self):
        return self.tab == []
    def empiler(self,elt):
        self.tab.append(elt)
    def depiler(self):
        return self.tab.pop()
```

9.3 Utilité

Lors du parcours d'un graphe en profondeur, les sommets à examiner sont stockés dans une pile.
Les appels récursifs d'une fonction utilisent une pile.

Chapitre 10

Jointures

10.1 Deux tables

10.1.1 Exemple

Pour modéliser un jeu de cartes, on propose ce schéma :

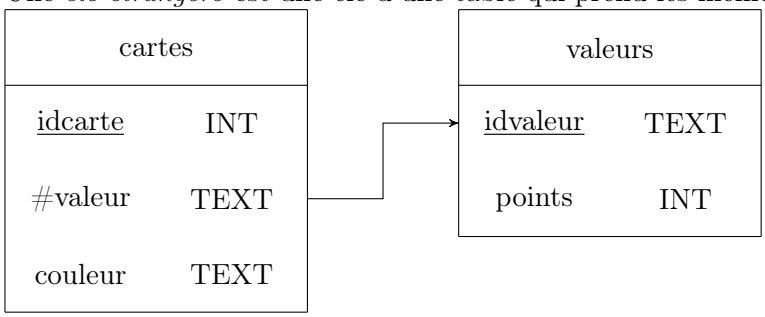
```
CREATE TABLE cartes (idcarte INT PRIMARY KEY,
                     valeur TEXT FOREIGN KEY REFERENCES valeurs(idvaleur),
                     couleur TEXT);
```

On entre les 52 cartes par

```
INSERT INTO cartes VALUES (1,'as','carreau');
INSERT INTO cartes VALUES (2,'2','carreau');
...
INSERT INTO cartes VALUES (13,'roi','carreau');
INSERT INTO cartes VALUES (14,'as','coeur');
...
INSERT INTO cartes VALUES (52,'roi','trefle');
```

10.1.2 Clé étrangère

Une *clé étrangère* est une clé d'une table qui prend les mêmes valeurs qu'une clé primaire d'une autre table.



10.1.3 contrainte de référence

```
INSERT INTO cartes VALUES (53,'cavalier','pique');
```

ne respecte pas la contrainte de clé étrangère : la clé primaire `idvaleur` ne prend jamais la valeur `cavalier`.

10.2 Jointure

10.2.1 Définition

Une *jointure* est une opération qui, à partir de deux tables, en construit une nouvelle. Il faut que les deux tables soient liées par un couple (clé étrangère, clé primaire) et la jointure des deux tables assure l'égalité

entre ces deux clés.
Exemple : la jointure entre `cartes` et `valeurs` permet d'ajouter à chaque carte, le nombre de points qu'elle rapporte.

10.2.2 SQL

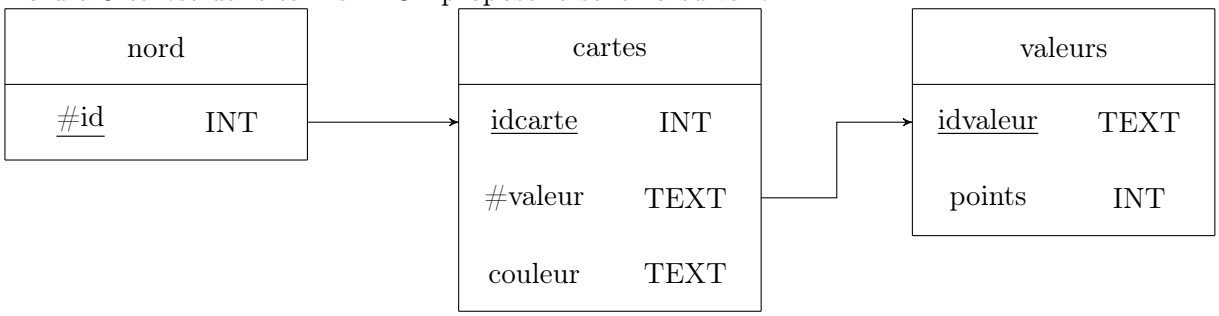
```
cartes JOIN valeurs
        ON valeur=idvaleur
```

10.2.3 Combien de points la 42ème carte rapporte-t-elle ?

```
SELECT points
        FROM cartes JOIN valeurs
                ON valeur=idvaleur
        WHERE idcarte=42;
```

10.3 Nord

Nord a 5 cartes dans sa main. On propose le schéma suivant :




```
CREATE TABLE nord (id INT PRIMARY KEY);
```

On distribue les cartes ainsi :

```
INSERT INTO nord VALUES (34),(11),(27),(37),(43);
```

Chapitre 11

Arbres binaires de recherche

11.1 Définition

Un *arbre binaire de recherche* est un arbre binaire dont les nœuds portent des nombres et

- tous les nœuds qui sont dans le sous-arbre gauche portent des valeurs inférieures à celle du nœud courant,
- tous les nœuds qui sont dans le sous-arbre droit portent des valeurs qui sont supérieures à celle du nœud courant.

11.2 Recherche

```
    def dedans(self,clef):
        if self.label==clef:
            return True
        elif self.label>clef and self.G is not None:
            return self.G.dedans(clef)
        elif self.label<clef and self.D is not None:
            return self.D.dedans(clef)
        else:
            return False
```

11.3 Insertion

```
    def ajouter(self,clef):
        assert not self.dedans(clef)
        if self.label>clef:
            if self.G is None:
                self.greffe_gauche(BST(clef))
            else:
                self.G.ajouter(clef)
        elif self.label<clef:
            if self.D is None:
                self.greffe_droite(BST(clef))
            else:
                self.D.ajouter(clef)
        else:
            self=BST(clef)
```

Chapitre 12

Diviser pour régner

12.1 Définition

La méthode *diviser pour régner* est une méthode de conception d'algorithmes, consistant à

- diviser le problème en problèmes analogues mais sur des données plus petites,
- résoudre (récursivement) ces problèmes
- recomposer la solution du problème initial à partir des solutions des problèmes intermédiaires.

Les algorithmes de ce type sont plus efficaces lorsque les données plus petites sont disjointes.

12.2 Recherche dichotomique dans une liste triée

```
def contient(tab,elt):
    n = len(tab)
    if n == 0:
        return False
    elif n == 1:
        return tab == [elt]
    else:
        return contient(tab[:n//2],elt) or contient(tab[n//2:],elt)
```

Cet algorithme est de coût logarithmique.

12.3 Tri fusion

(Von Neumann, Goldstine 1947)

12.3.1 fusion

```
def fusion(liste1: list, liste2:list)->list:
    if liste1 == []:
        return liste2
    elif liste2 == []:
        return liste1
    else:
        a = liste1[0]
        b = liste2[0]
        if a<=b:
            return [a] + fusion(liste1[1:], liste2)
        else:
            return [b] + fusion(liste1,liste2[1:])
```

- Un variant est la somme des longueurs des deux listes à fusionner : à chaque appel récursif elle décroît d'une unité.

- Un invariant est *si les deux listes sont triées dans l'ordre croissant alors la liste fusionnée aussi.*
- L'algorithme est de coût linéaire.

12.3.2 Tri fusion

```
def mergesorted(liste: list)->list:
    n = len(liste)
    if n <= 1:
        return liste
    else:
        return fusion(mergesorted(liste[:n//2]),mergesorted(liste[n//2:]))
```

12.3.3 Propriétés

- La longueur de la liste à trier est un variant : elle est divisée par 2 à chaque appel récursif.
- Un invariant est *les deux listes à fusionner sont triées.*
- Le coût de cet algorithme est en $n \times log_2(n)$.

Chapitre 13

Files

13.1 Définitions

13.1.1 File

Une *file* est une liste accessible à un bout pour l'ajout et à un autre bout pour le retrait.

13.1.2 FIFO

First In, First Out décrit le comportement d'une file.

13.2 Classe File

```
class File:
    def __init__(self):
        self.tab = []
    def est_vide(self):
        return self.tab == []
    def enfiler(self,elt):
        self.tab.append(elt)
    def defiler(self):
        return self.tab.pop(0)
```

13.3 Usage des piles

```
class File:
    def __init__(self):
        self.pile1 = Pile()
        self.pile2 = Pile()
    def est_vide(self):
        return self.pile1.est_vide() and self.pile2.est_vide()
    def flush(self):
        while not self.pile1.est_vide():
            self.pile2.empiler(self.pile1.depiler())
    def enfiler(self,elt):
        self.pile1.empiler(elt)
    def defiler(self):
        if self.pile2.est_vide():
            self.flush()
        return self.pile2.depiler()
```

Chapitre 14

Cryptographie

14.1 Chiffrement symétrique

```
def chiffrement(message:str, clef:list)->str:
        message_code = ''
        position  = 0
        for lettre in message:
                message_code += chr(ord(lettre)^clef[position])
                position = (position+1)%len(clef)
        return message_code
```

Avantages du chiffrement symétrique :

- La même clé sert au chiffrement et au déchiffrement.
- utilisable pour tout fichier binaire
- rapide (à la volée)

Inconvénient : il faut partager la clé préalablement.

14.2 Chiffrement asymétrique

Dans un chiffrement comme RSA, il y a deux clés :

- une clé de chiffrement (clé publique),
- une clé de déchiffrement (clé privée)

Avantage : seule la personne qui dispose de la clé privée peut déchiffrer le message.
Inconvénients :

- Les calculs arithmétiques sont lents (élévation à une puissance modulo un grand entier)
- consomme de l'énergie
- On ne peut garantir en théorie que c'est incassable

14.3 Partage de clé symétrique

(Whit Diffie, Martin Hellman 1976)
La clé symétrique ne doit être connue que d'Alice et Bobo, c'est leur *secret*.

- Alice crypte le secret avec la clé de chiffrement de Bob.
- Alice envoie le secret cryptée à Bob.
- Bob crypte le secret crypté avec la clé de chiffrement d'Alice.
- Bob envoie à Alice le secret surcrypté.
- Alice décrypte avec sa clé de déchiffrage.
- Alice envoie à Bob le secret (crypté avec la clé de chiffrement de Bob)
- Bob décrypte avec sa clé de déchiffrement la clé crypté, et obtient le secret.

HTTPS échange les clés symétriques (WEP, WPA, ...) avec RSA.

Chapitre 15

Programmation dynamique

15.1 Position du problème

L'algorithme glouton de rendu de monnaie :

```
def monnaie(n:int)->dict:
    bourse = {1:0, 3:0, 4:0}
    while n>=4:
        n -= 4
        bourse[4] += 1
    while n>=3:
        n -= 3
        bourse[3] += 1
    while n>=1:
        n -= 1
        bourse[1] += 1
    return bourse
```

n'est pas optimal. Si on ajoute une fonction

```
def poids(dico:dict)->int:
    return sum(dico.values())
```

on constate que `poids({1:2,3:0,4:1})==3` alors que `poids({1:0,3:2,4:0})==2` seulement.

15.2 Solution récursive

```
def monnaie(n:int)->dict:
    if n<3:
        return {1:n,3:0,4:0}
    elif n<4:
        return {1:0,3:1,4:0}
    else:
        d1 = monnaie(n-1)
        d3 = monnaie(n-3)
        d4 = monnaie(n-4)
        if poids(d4)<poids(d3):
            d4[4] += 1
            return d4
        elif poids(d3)<poids(d1):
            d3[3] += 1
            return d3
        else:
```

```
            d1[1] += 1
            return d1
```

15.3 Programmation dynamique

```
def monnaie(n):
    vu = []
    for k in range(n+1):
        if k<3:
            vu += [(k,0,0)]
        elif k<4:
            vu += [(0,1,0)]
        else:
            t1 = vu[k-1]
            t3 = vu[k-3]
            t4 = vu[k-4]
            if sum(t4)<sum(t3):
                vu.append((t4[0],t4[1],t4[2]+1))
            elif sum(t3)<sum(t1):
                vu.append((t3[0],t3[1]+1,t3[2]))
            else:
                vu.append((t1[0]+1,t1[1],t1[2]))
    return vu[-1]
```

Chapitre 16

Calculabilité et décidabilité

16.1 Terminaison

```
I = 'x=0\nwhile x<8:\n    x+=1'
exec(I)
```

se termine (variant : `8-x`) avec `x == 8`

```
I = 'x=1\nwhile x>0:\n    x+=1'
exec(I)
```

ne se termine pas.
On s'intéresse à une fonction Python

```
def se_termine(programme: string) -> bool:
    ....
    if ....:
        return True
    else:
        return False
```

16.2 Théorème de Turing

(Alan Turing 1936)

```
def T(programme):
    if se_termine(programme):
        x = 1
        while x>0:
            x += 1
    else:
        return True
```

La chaîne de caractères

```
programmeT = "T(" + P + ")"
```

est telle que

```
exec(programmeT)
```

se termine uniquement quand `P` ne se termine pas. Donc `T('T(programmeT)')` se termine uniquement si elle ne se termine pas.
Il n'existe pas de fonction Python qui puisse décider si un programme Python se termine.