Saint graal NSI

Ce document traite en majorité des plus grandes difficultés dans le cours de NSI, les parties les plus basiques sont ignorées puisque l'épreuve est dans - de 17h.

Orienté objet (le fameux)

Exemple présenté avec des voitures

```
class Voiture:
    def __init__(self, marque, modele): # constructeur
        self.marque = marque # marque est un attribut
        self.modele = modele

    def afficher_details(self): # c'est une méthode
        print(f"Marque: {self.marque}, Modèle: {self.modele}")

# Création d'un objet
ma_voiture = Voiture("Toyota", "Corolla")
```

```
# Utilisation des attributs
print(ma_voiture.marque)  # Toyota
print(ma_voiture.modele)  # Corolla

# Utilisation d'une méthode
ma_voiture.afficher_details()  # Marque:
Toyota, Modèle: Corolla
```

Pile et file

Ce sont des structures de données linéaire.

Pile

Une pile est une structure de données dans laquelle les éléments sont ajoutés et supprimés selon le principe LIFO (Last In, First Out), c'est-à-dire que le dernier élément ajouté est le premier à être retiré.

```
class Pile:
    def __init__(self):
        self.elements = []

    def empiler(self, element):
        self.elements.append(element)

    def depiler(self):
        if not self.est_vide():
```

```
            return self.elements.pop()
        else:
            return "La pile est vide"

    def sommet(self):
        if not self.est_vide():
            return self.elements[-1]
        else:
            return "La pile est vide"

    def est_vide(self):
        return len(self.elements) == 0

# Utilisation de la pile
pile = Pile()
pile.empiler(1)
pile.empiler(2)
pile.empiler(3)
print(pile.depiler())  # 3
print(pile.sommet())  # 2
print(pile.est_vide())  # False
```

File

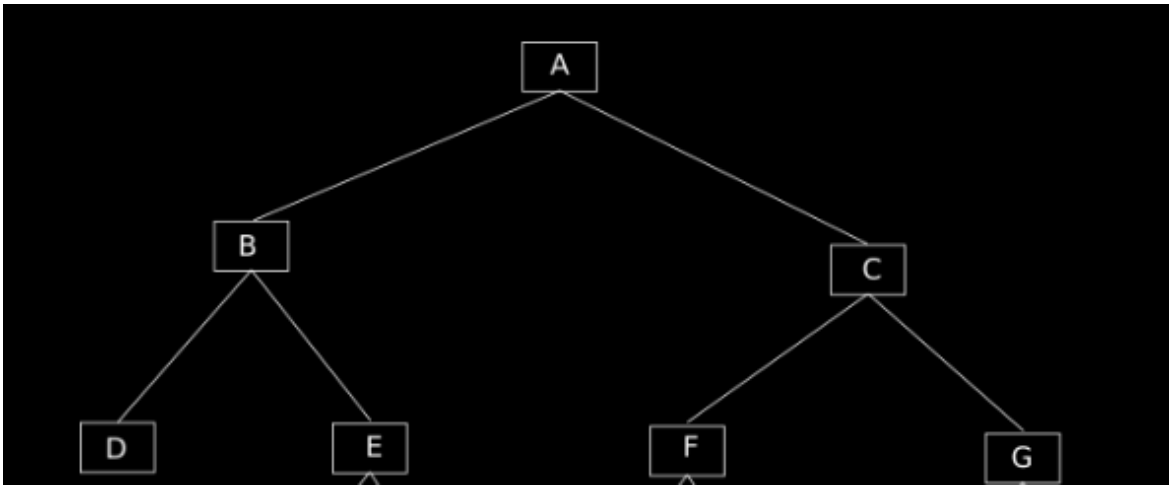
Une file est une structure de données dans laquelle les éléments sont ajoutés à une extrémité (arrière) et retirés de l'autre (avant), suivant le principe FIFO (First In, First Out), c'est-à-dire que le premier élément ajouté est le premier à être retiré.

```python
class File:
    def __init__(self):
        self.elements = []

    def enfiler(self, element):
        self.elements.append(element)

    def defiler(self):
        if not self.est_vide():
            return self.elements.pop(0)
        else:
            return "La file est vide"

    def premier(self):
        if not self.est_vide():
            return self.elements[0]
        else:
            return "La file est vide"

    def est_vide(self):
        return len(self.elements) == 0

# Utilisation de la file
file = File()
file.enfiler(1)
file.enfiler(2)
file.enfiler(3)
print(file.defiler())  # 1
print(file.premier())  # 2
print(file.est_vide())  # False
```

Arbre binaire

Arbre binaire : structure de données non linéaire. Ce qui fait que c'est un arbre binaire c'est que chaque branche a au max 2 enfants.

Definitions



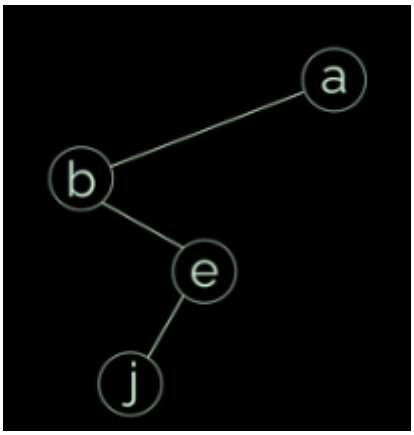
- noeud racine : A
- relation parent/enfant : B parent de E, C enfant de A
- feuille : noeud sans enfant
- On peut définir un sous arbre, exemple le sous arbre droit du noeud A est l'arbre C,F,G
- arrete : segment qui relie 2 noeuds
- chemin : ensemble d'arretes reliant 2 noeuds

- branche : chemin plus court reliant noeud a la racine
- profondeur : nombre d'arrete de la branche/nombre de segment le reliant au noeud racine
- hauteur : profondeur maximal de l'arbre
- taille : nombre de noeud
- arite d'un noeud : nombre d'enfant
- arité d'un arbre : nombre max d'enfant dans tout l'arbre

Types d'arbres

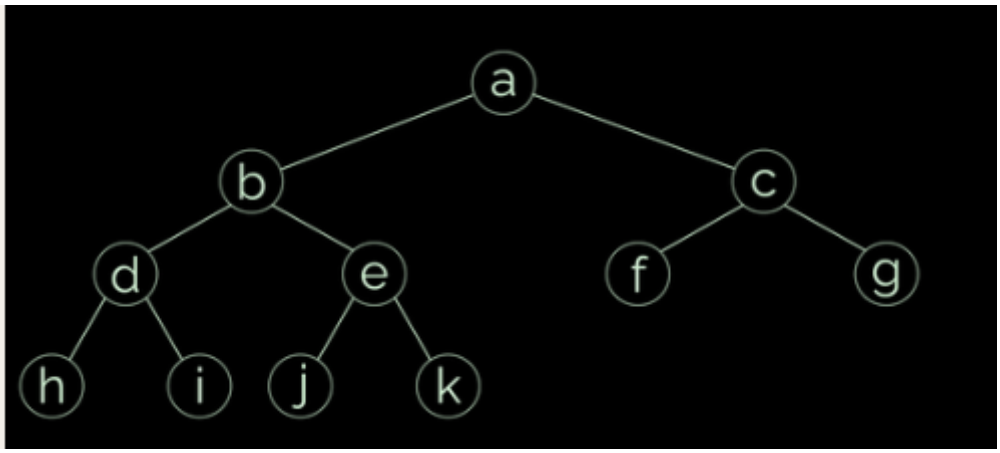
Filiforme ou dégénéré

arité des noeuds <= 1



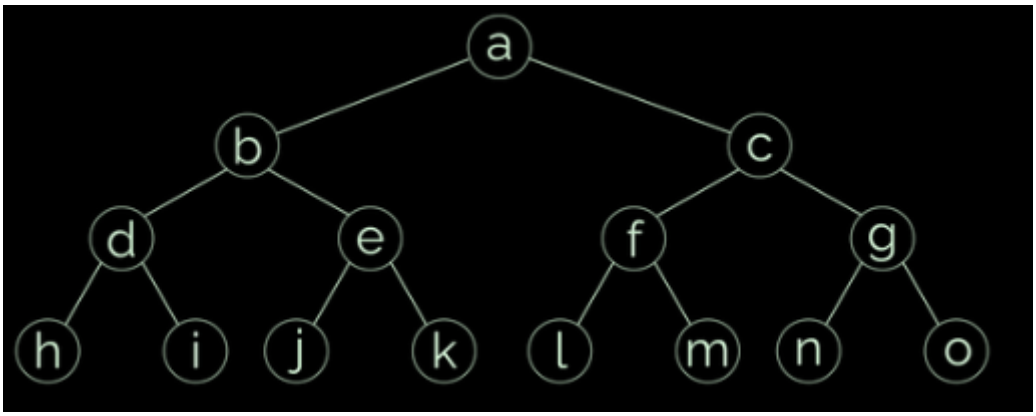
Localement complet/strict

arité des noeuds = 2 ou 0 pour les feuilles



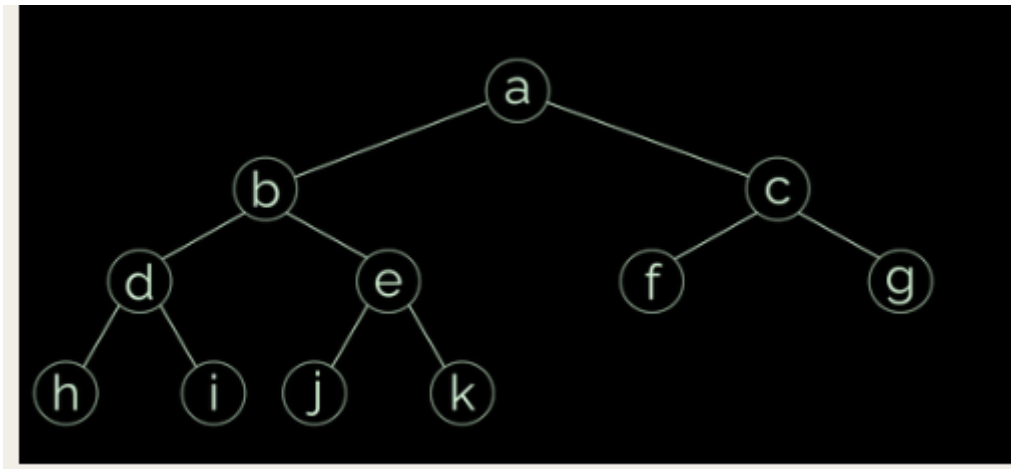
Complet

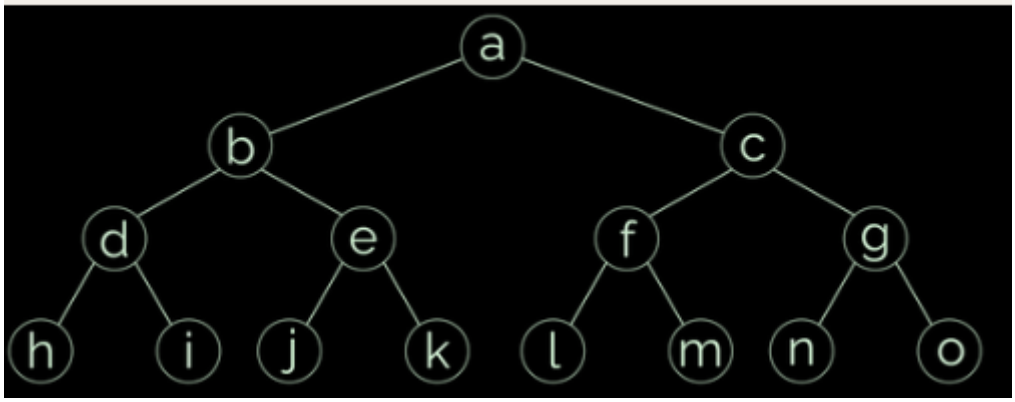
Toutes les feuilles sont a la meme profondeur



Equilibré

hauteur du sous arbre droit et gauche ont la meme hauteur a + ou - 1





Algorithmes

Definition de l'arbre binaire en orienté objet

> **Note**
>
> Ici, tous les exemples présenté sont en orienté objet, mais des variants + classiques existent

```
class ArbreBinaire:
    def __init__(self,etiquette=None,sag=None,
sad=None): #Creer la structure, avec une
```

```
étiquette (la valeur du noeud (cela peux etre
un nombre ou une lettre, voir une chaine de
caractere (strin)), son sous arbre gauche et
son sous arbre droit qui sont des arbres
binaires)
        self.etiquette=etiquette
        self.sag=sag
        self.sad=sad

    def est_vide(self): #Renvoie True si
l'arbre est vide, sinon False
        if self.etiquette is None:
            return True
        else:
            return False
```

Taille de l'arbre

```
    def taille(self):
        if self.est_vide():
            return 0
        else:
            return 1 + self.sag.taille() +
self.sad.taille()
```

Hauteur de l'arbre

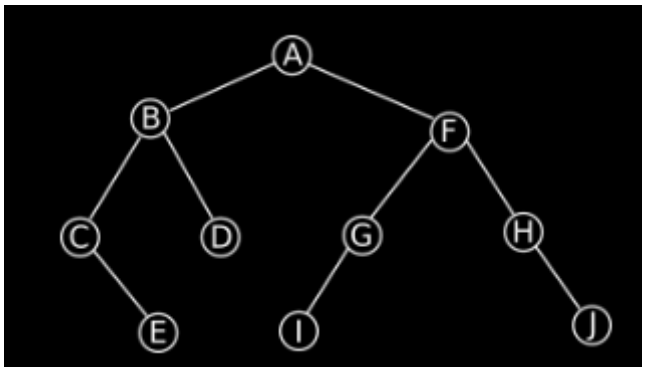
Pour calculer la hauteur d'un arbre binaire, nous pouvons utiliser un algorithme par récurrence

> ⚡ **Attention**
>
> ATTENTION, dans la condition d'arrêt il faut "return -1", car l'algorithme compte les sous arbres vides

```
def hauteur(self):
        if self.est_vide():
                return -1
        else:
                return 1 +
max(self.sag.hauteur(), self.sad.hauteur())
```

Parcours

Tous les parcours seront appliqué a cette arbre binaire



Infixe

```
def parcoursInfixe(self):
   if not self.etiquette is None:
                x = self
        self.sag.parcoursInfixe()
```

```
        print(x.etiquette)
        self.sad.parcoursInfixe()
```

Le parcours de cette arbre donne : C, E, B, D, A, I, G, F, H, J

Suffixe / postfixe

```
def parcoursSuffixe(self):
    if not self.etiquette is None:
        x = self
        self.sag.parcoursSuffixe()
        self.sad.parcoursSuffixe()
        print(x.etiquette)
```

Le parcours de cette arbre donne : E, C, D, B, I, G, J, H, F, A

Préfixe

```
def parcoursPrefixe(self):
    if not self.etiquette is None:
        x = self
        print(x.etiquette)
        self.sag.parcoursPrefixe()
        self.sad.parcoursPrefixe()
```

Le parcours de cette arbre donne : A, B, C, E, D, F, G, I, H, J

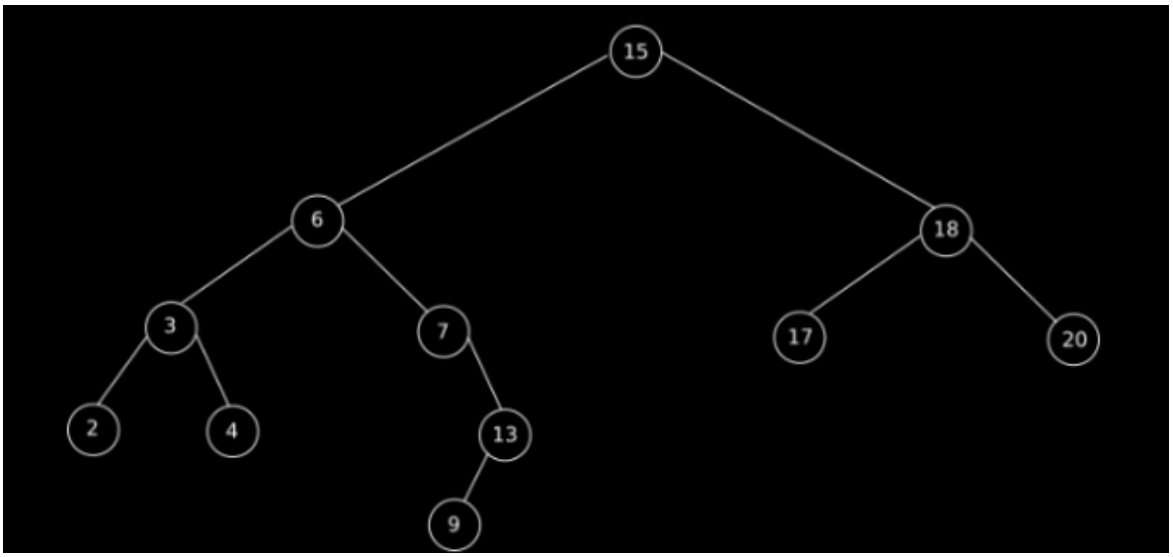
Parcours en largeur

```
def parcoursLargeur(self):
    f = File()
    f.enfiler(self)
    while f.est_vide() == False:
        x = f.defiler()
        print(x.etiquette)
        if not x.sag.etiquette is None:
            f.enfiler(x.sag)
        if not x.sad.etiquette is None:
            f.enfiler(x.sa)
```

Le parcours de cette arbre donne : A, B, F, C, D, G, H, E, I, J

Arbre binaire de recherche

Arbre binaire de recherche



Critères pour que ca soit un arbre binaire de recherche

- Doit être un arbre binaire (vu précédemment)
- Les clés doivent être ordonnable (nombre : croissant, lettre : alphabet)
- Les clés des sous arbre droites doivent être plus grand que la clé du noeud parent
- Les clés des sous arbres gauches doivent être plus petite que la clé du noeud parent

Algorithme de recherche

```
def trouver(self, el): #Renvoie true si l'élement est présent et False s'il n'est pas présent dans l'arbre. el = element qu'on recherche dans l'arbre
    if self.etiquette is None:
        return False
    x = self
    if x.etiquette == el:
        return True
    if el < x.etiquette:
        return self.sag.trouver(el)
```

```
    else:
        return self.sad.trouver(el)
```

L'algorithmes de recherche a un cout logarithmique : n * log(n)

Algorithme d'insertion

```
def inserer(self, cle):
    if self.est_vide():
        return ArbreBinaire(cle, None, None)
    if cle < self.etiquette:
        return ArbreBinaire(self.etiquette,
self.sag.inserer(cle), self.sad)
    else:
        return ArbreBinaire(self.etiquette,
self.sag, self.sad.inserer(cle))
```

Graphes (fun :[)

Définition

- Ordre d'un graphe : nombre de sommets.
- Adjacence : sommet t adjacent à sommet s s'il y a une arête de s vers t.
- Degré d'un sommet : nombre d'arêtes issues de ce sommet (entrant et sortant pour un graphe orienté).
- Sommet isolé : sommet sans adjacence.

- Graphe complet : tous les sommets distincts sont adjacents.
- Chaîne (ou chemin) : suite d'arêtes consécutives.
- Longueur d'une chaîne : nombre d'arêtes dans la chaîne.
- Distance entre deux sommets : longueur de la plus petite chaîne qui les relie.
- Centre d'un graphe : sommet avec la distance la plus petite aux autres sommets (un graphe peut avoir plusieurs centres).
- Rayon d'un graphe : plus petite distance reliant le centre à tous les autres sommets.
- Diamètre d'un graphe : plus grande distance possible entre deux sommets.
- Chaîne fermée : chaîne dont l'origine et l'extrémité coïncident.
- Cycle : chaîne fermée avec des arêtes distinctes.
- Graphe connexe : dans un graphe non orienté, il existe une chaîne entre chaque paire de sommets.
- Graphe fortement connexe : dans un graphe orienté, il existe un chemin de

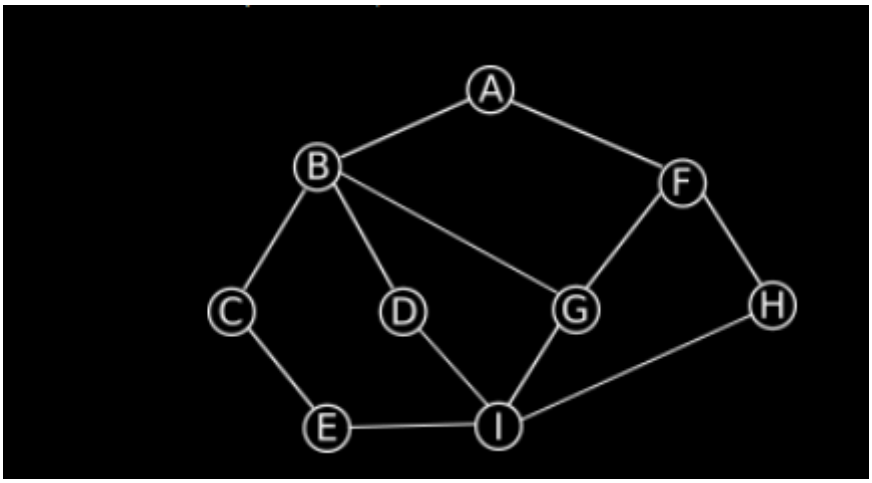
u vers v et de v vers u pour chaque paire de sommets distincts (u, v).

Algo

Parcours en largeur

Ce parcours fonctionne à l'aide d'une pile, on commence a un point, on met tous les sommets adjacents dans cette file, on prend le 1er sommet dans cette pile et on met tous ces enfants dans la file si on ne les a pas encore visité, il se termine une fois qu'on a parcourus tous les sommets (il existe plusieurs parcours en largeur selon l'ordre où l'on met les sommets dans la file)

CAD -> on fait tous les points par ordre de distance avec le point de depart



parcours : A, B, F, C, D, G, H, E et I (pas la seule solution possible)

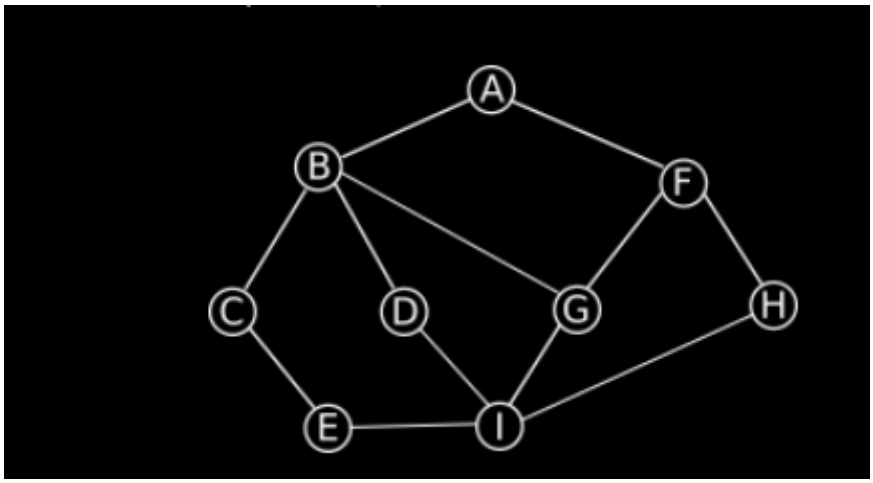
```
# version itérative
VARIABLE
G : un graphe
s_depart : sommet (origine)
s : sommet
n : sommet
f : file (initialement vide)
visite : liste des sommets visités
(initialement vide)

DEBUT
enfiler (s_depart,f)
tant que f non vide :
  s ← defiler(f)
  si s n'appartient pas à visite :
    ajouter s à visite
    pour chaque sommet n adjacent au sommet s
:
      si n n'appartient pas à visite :
        enfiler(n,f)
      fin si
    fin pour
  fin si
fin tant que
renvoyer visite
FIN

# version récursive
VARIABLE
G : un graphe
s_depart : sommet (origine)
s : sommet
```

```
visite : liste des sommets visités
(initialement vide)

DEBUT
PARCOURS-PROFONDEUR(G,s_depart,visite) :
  si s_depart n'appartient pas à visite :
    ajouter s_depart à visite
  fin si
  pour chaque sommet s adjacent au sommet
s_depart :
    si s n'appartient pas à visite :
      PARCOURS-PROFONDEUR(G,s,visite)
    fin si
  fin pour
  renvoyer visite
FIN
```

Parcours en profondeur

Pareil que le parcours en largeur mais cette fois on utilise une pile.

CAD -> on va le plus loin possible dans notre recherche, une fois bloqué, on remonte où on a pas encore été




```
# Version itérative
pareil que le parcours en largeur mais avec
une pile

# Version récursive
VARIABLE
G : un graphe
s_depart : sommet (origine)
s : sommet
visite : liste des sommets visités
(initialement vide)

DEBUT
PARCOURS-PROFONDEUR(G,s_depart,visite) :
  si s_depart n'appartient pas à visite :
    ajouter s_depart à visite
  fin si
  pour chaque sommet s adjacent au sommet
s_depart :
    si s n'appartient pas à visite :
      PARCOURS-PROFONDEUR(G,s,visite)
    fin si
  fin pour
```

```
  renvoyer visite
FIN
```

Recherche dans un graph

Algorithme pour recherche une chaine dans un graphe entre 2 sommet

```
VARIABLE
G : un graphe
s_depart : sommet (sommet de départ)
s_fin : sommet (sommet d'arrivée)
s : sommet
f : file (initialement vide)
chemins : liste des chemins possibles entre
s_depart et s_fin (initialement vide)
sommet : sommet
chemin : liste de sommets d'un chemin possible

DEBUT
enfiler ((s_depart,[s_depart]),f)
tant que f non vide :
  sommet,chemin ← defiler(f)            # f
est une file de tuples!
  pour chaque sommet s adjacent à sommet :
    si s n'appartient pas à chemin :
          si s est le sommet d'arrivée :
                ajouter chemin + [s] à chemins
          sinon :
                enfiler ((s,chemin + [s]),f)
          fin si
```

```
    fin si
  fin pour
fin tant que
renvoyer chemins
FIN
```

Recherche de cycle

```
VARIABLE
G : un graphe
s_depart : sommet (sommet de départ)
s : sommet
f : file (initialement vide)
cycles : liste des cycles possibles à partir
de s_depart (initialement vide)
sommet : sommet
chemin : liste de sommets d'un chemin possible

DEBUT
enfiler ((s_depart,[s_depart]),f)
tant que f non vide :
  sommet,chemin ← defiler(f)            # f
est une file de tuples!
  pour chaque sommet s adjacent à sommet :
    si s n'appartient pas à chemin ou si s est
le sommet de départ :
          si s est le sommet de départ :
                ajouter chemin + [s] à cycles
          sinon :
                enfiler ((s,chemin + [s]),f)
          fin si
```

```
    fin si
  fin pour
fin tant que
renvoyer cycles
FIN
```

Base de donnée

Dans une base de données, l'information est stockée dans des fichiers. Contrairement aux fichiers CSV, on ne peut pas les éditer avec un simple éditeur de texte.

Pour gérer les données (écrire, lire, modifier), on utilise un logiciel appelé "système de gestion de base de données" ou SGBD.

Fonctions des SGBD

1. Gestion des données : Les SGBD permettent de lire, écrire et modifier les données dans une base de données.
2. Gestion des accès : Ils contrôlent qui peut lire ou écrire dans la base de données, en donnant différents niveaux d'accès aux utilisateurs.
3. Fiabilité et redondance : Les bases de données sont souvent dupliquées sur plusieurs ordinateurs pour assurer

l'accès aux données même en cas de panne. Les SGBD synchronisent les modifications entre ces copies pour qu'elles restent identiques.

4. Accès concurrent : Les SGBD gèrent les accès simultanés aux données, évitant les conflits lorsque plusieurs utilisateurs modifient la même donnée en même temps.

Les SGBD simplifient la gestion des bases de données et permettent de surpasser les solutions plus simples comme les fichiers CSV.

SQL

Définition

- Clé primaire : numéros unique qui permet que chaque enregistrement soit unique.
- Clé étrangère : Clé primaire d'un autre tableau pour établir une relation
- Base de donnée relationnelle : base de donnée regroupant plusieurs tables qui sont mises en relation grace a des jointures

- Bonne base de donnée : Si il y a besoin de ne faire uniquement 1 seule modification a faire pour changer une valeur dans la base de donnée.
- Pour gerer une base de donnée il faut un logiciel de type SGBD (Systeme de Gestion de Base de Donnée)

Les instructions

L'instruction 'SELECT'

Cette instrucion permet de définir quelles sont les clé choisit à afficher.

- Pour selectionner toutes les données d'une base de donnée

```
SELECT * FROM ville;
```

- Pour selectionner certaines clés données d'une base de donnée

```
SELECT nom, code FROM ville;
```

- Pour renomer une clé d'une base de donnée

```
 SELECT nom AS NOM_VILLE FROM ville;
```

- Pour éliminer les doublons

```
SELECT DISTINCT code FROM ville;
```

- Pour trier les éléments

```
SELECT code FROM ville ORDER BY code;
```

- Pour inverser le sens de tri, on peut utiliser 'DESC'

```
SELECT code FROM ville ORDER BY code DESC;
```

L'instruction 'WHERE'

Cette instruction permet de faire un filtrer et de sélectionner des élements uniquement si une condition est validé.

```
SELECT * FROM ville WHERE code='59140';
#Selectionne tous les élements où le code =
'59140
SELECT nom, code FROM ville WHERE
code='67340';
#Selectionne le nom et le code où le code =
'59140
SELECT nom, code FROM ville WHERE code='59140'
OR code='59260';
```

L'instruction 'INSERT'

Permet d'ajouter des données dans une table.

```
INSERT INTO ville VALUES
('62000','31','62','Calais','50.291048,2.77722
11');
```

> ATTENTION : on obtient une erreur lorsque l'on essaye d'ajouter une des données qui sont deja présent.

L'instruction 'UPDATE'

Permet de mettre a jour des données deja présente dans la table.

```
UPDATE ville SET code='62001' WHERE
code='62000';
#Remplace dans la table ville, le code par
62001, où le code = 62000
```

L'instruction 'DELETE'

Permet de supprimer des données présente dans la table.

```
DELETE FROM ville WHERE code='62001';
```

Instruction supplémentaire

> Ces instructions seront sûrement donnée, si elle sont nécessaire à la réalisation de l'exercice lors du BAC. De ce fait, elles ne sont pas très importante mais les connaîtres est toujours un plus.

- Pour avoir a plus petite ou la plus grande valeur

```
SELECT MIN(code) AS MIN FROM ville;
SELECT MAX(code) AS MAX FROM ville;
# Pour avoir le nombre de ligne du tableau
SELECT COUNT(*) AS NbLignes FROM ville;
# Pour avoir la somme
SELECT SUM(effectif) AS Somme_effectif FROM
ville;
# Pour avoir la moyenne
SELECT AVG(effectif) AS moyenne FROM
evolution;
```

Croisement de plusieurs tables

Croiser des tables permet de ne pas avoir des données dupliqué, exemple : ici il y a 2 tables, une premiere table *ville* qui contient un code unique pour chaque vielle ainsi que les noms des villes, leur

position géographique, ect... et une autre table *evolution* qui elle contient aussi le meme code, le nom du métier et l'éffectif du métier.

Pour joindre 2 table on peut utiliser l'instruction 'JOIN'

```
SELECT ville.nom, evolution.effectif FROM
ville JOIN evolution ON ville.code =
evolution.code;
```

Les autres instructions fonctionnent aussi :

```
SELECT ville.nom, evolution.effectif FROM
ville JOIN evolution ON ville.code =
evolution.code WHERE evolution.effectif > 2000
ORDER BY evolution.effectif;
```

Commandes linux

1. cd (Change Directory)
    - Description : Change le répertoire courant.
    - Syntaxe : `cd [répertoire]`
    - Exemples :

```
cd /home/user      # Aller au
répertoire /home/user
cd ..              # Aller au
répertoire parent
cd                 # Aller au
répertoire personnel de l'utilisateur
```

2. **pwd (Print Working Directory)**
    - **Description :** Affiche le chemin du répertoire courant.
    - **Syntaxe :** `pwd`
    - **Exemple :**

```
pwd                # Affiche le chemin
complet du répertoire courant
```

3. **man (Manual)**
    - **Description :** Affiche le manuel d'utilisation des commandes.
    - **Syntaxe :** `man [commande]`
    - **Exemple :**

```
man ls             # Affiche le manuel
de la commande ls
```

4. **ls (List)**
    - **Description :** Liste les fichiers et répertoires.

- Syntaxe : `ls [options] [chemin]`
- Options Utiles :
  - `-l` : Affiche les détails des fichiers.
- Exemples :

```
ls                  # Liste les
fichiers dans le répertoire courant
ls /etc             # Liste les
fichiers dans le répertoire /etc
```

5. mkdir (Make Directory)
   - Description : Crée un nouveau répertoire.
   - Syntaxe : `mkdir répertoire`
   - Exemples :

```
mkdir mon_repertoire           # Crée
un répertoire nommé "mon_repertoire"
```

6. id
   - Description : Affiche les informations sur l'utilisateur et le groupe.
   - Syntaxe : `id [utilisateur]`
   - Exemples :

```
id                  # Affiche les
informations sur l'utilisateur courant
id utilisateur      # Affiche les
informations sur "utilisateur"
```

Protocoles internet

- Adresse IP : Une adresse IP est un numéro unique qui identifie un appareil connecté à un réseau. C'est comme une adresse postale pour les ordinateurs. ex : *172.168.5.2*
- Adresse MAC : Identifiant unique attribué a chaque carte réseau, (un peu comme le numéros de série pour la carte réseau)
- Réseaux locaux : Les réseaux locaux (LAN) sont des réseaux informatiques utilisés dans des endroits proches les uns des autres, comme une *maison* ou un *bureau*, pour que les appareils puissent se connecter et communiquer entre eux.
- Switch : Un switch est un appareil qui connecte plusieurs appareils dans un réseau local. Il dirige les données uniquement vers l'appareil

destinataire, ce qui rend les communications plus efficaces.

- Routeur :Un routeur est un appareil qui dirige le trafic réseau entre différents réseaux. Il s'assure que les données parviennent de l'expéditeur au destinataire correctement.
- Masque de sous-réseau : Le masque de sous-réseau est une série de chiffres utilisée pour diviser un réseau en sous-réseaux plus petits. Cela permet d'organiser le trafic et de limiter la visibilité des appareils dans le réseau. ex: *255.255.255.0*

Attention le masque peut prendre d'autres formes ex : *255.240.00*

- Paquet : Un paquet est une petite quantité de données envoyée sur un réseau informatique. C'est comme un petit colis qui transporte des informations d'un endroit à un autre sur Internet. Chaque paquet contient les données à envoyer, ainsi que des instructions sur la manière de les livrer à leur destination.

Masques de sous réseaux

Prenons l'exemple d'un appareil placé connecté a internet, avec l'adresse IP : 192.168.1.5, et un masque de sous réseau 255.255.255.0

Ce masque sert à avoir l'adresse du réseau. Pour récupérer l'adresse du réseaux, il faut d'abord convertir l'adresse IP et le masque en binaire Conversion Binaire.

192.168.1.5 :
11000000.10101000.00000001.00000101
255.255.255.0 :
11111111.11111111.11111111.00000000
Il suffit ensuite d'appliquer la Porte ET a ces 2 nombres

11000000.10101000.00000001.00000101
11111111.11111111.11111111.00000000
=
11000000.10101000.00000001.00000000

On reconvertit en nombre entier
Adresse réseau : 192.168.1.0

Trouver le nombre d'appareil connectable

On peut trouver le nombre d'appareil maximum qu'on peut connecté un réseau en

meme temps, a l'aide du masque de sous réseau

Il faut compter combien de bit sont a 0.

> **Note**
>
> Les adresses ip 255.255.255.255 et 255.255.255.0 sont reservé, alors il faut enlever 2 au résultat.

Ex : 255.255.255.0
il y a 8 bit a 0, donc $2^8 - 2$, soit $256 - 2 = 254$ réseaux possible.

Ex : 255.255.0.0
il y a 16 bit a 0, donc $2^{16} - 2$ réseaux possible.

Protocoles internet

Pour envoyer des données, il faut trouver un itinéraire parmi cette toile de routeurs. C'est la que 2 protocoles rentrent en jeu : RIP, et OSPF

RIP

RIP = Routing Information Protocol
Fonctionne en comptant le nombre de saut entre la destination cible et le point de

depart, (max de 15 routeurs pour atteindre la destination).

OSPF

OPSF = Open Shortest Path First
Prend en compte la valeur de la bande passante, pour chaque saut, on calcule le cout, a l'aide de cette formule $cout = \frac{10^8}{debit}$
On prend le chemin ou le cumule des couts est le plus faible.

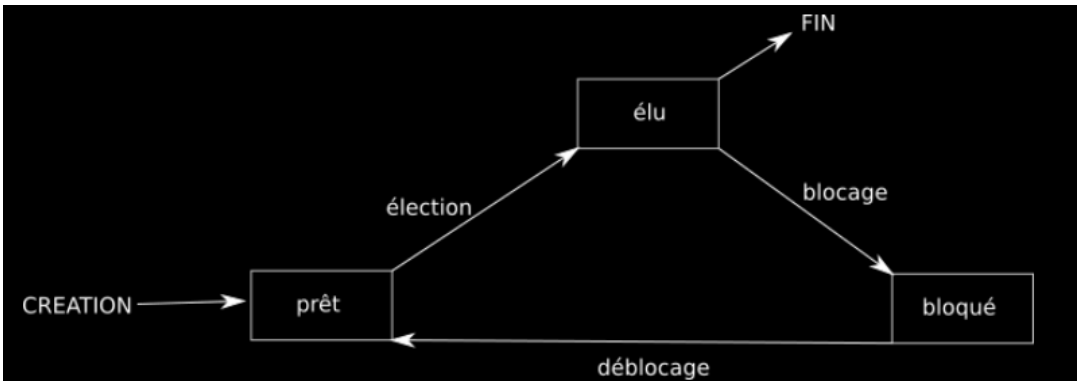
> ⚡ **ATTENTION**
>
> Selon les exercices, ca peut ce pas 10^8

Architecture materiel

Processus

- Chaque processus excepté le processus 0, provient d'un processus parent
- chaque processus a 2 identifiants : PID (Process Identification) identifiant du processus (unique)
- PPID (Parent Process Identification) qui est le processus parent qui a créé ce processus

- Un processus a plusieurs états :



Situation d'Interblocage/deadlock

C'est quand plusieurs processus sont bloqué puisqu'il demande tous une ressource deja bloqué.

Diviser pour reigner

Cette facon de coder, consiste a diviser un problème en pleins de petits problèmes plus facile a résoudre, puis de combiner le résultat
exemple avec le tri-fusion, qui divise une liste en pleins de liste de 2 ou 1 element/s qui sont plus facile a trier.

```
VARIABLE
tab : tableau d'entiers
tab1 : tableau d'entiers
tab2 : tableau d'entiers
tab_sortie : tableau d'entiers
```

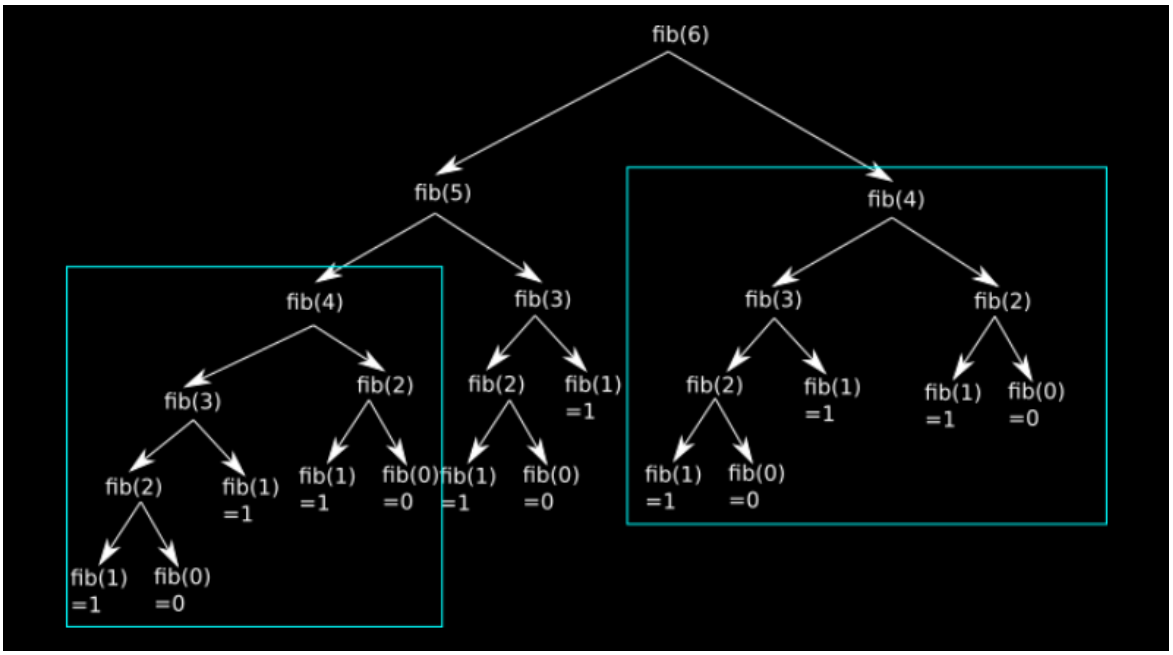
```
i1 : entier
i2 : entier
n : entier


DEBUT

FUSION (tab1, tab2):
  i1 ← 0
  i2 ← 0
  tab_sortie ← []
  tant que i1 < longueur de tab1 ET i2 <
longueur de tab2:
    si tab1[i1] < tab2[i2]:
      ajouter tab1[i1] à tab_sortie
      i1 ← i1+1
    sinon:
      ajouter tab2[i2] à tab_sortie
      i2 ← i2+1
    fin si
  fin tant que

  si i1 = longueur de tab1:
    ajouter le reste de tab2 à partir de
l'indice i2 à tab_sortie
  sinon:
    ajouter le reste de tab1 à partir de
l'indice i1 à tab_sortie
  fin si

  renvoyer tab_sortie
fin FUSION
```

```
TRI-FUSION(tab):
  n ← longueur de tab
  si n < 2:
    renvoyer tab
  sinon
    renvoyer FUSION(TRI-FUSION(n//2 premiers
éléments de tab),TRI-FUSION(n//2 derniers
éléments de tab))
  fin si
fin TRI-FUSION

FIN
```

Programmation dynamique

Avec des fonctions ou méthodes récursive, le meme calcul est fait plusieurs fois, dans cette facon de programmer on enregistre les anciens résultats afin de ne pas les calculer de nouveau.

exemple avec Fibonacci



Implémentation normal

```
def fib(n) :
  if n < 2 :
    return n
  else :
    return fib(n-1)+fib(n-2)
```

Implémentation dynamique

```
def fib_mem(n):
  mem = [None]*(n+1)  #permet de créer un
tableau contenant n+1 éléments None
  return fib_mem_c(n,mem)

def fib_mem_c(n,m):
  if m[n] is not None:
    return m[n]
```

```
  if n < 2 :
    m[n]=n
    return n
  else:
    m[n]=fib_mem_c(n-1,m) + fib_mem_c(n-2,m)
    return m[n]
```

Autre exemple avec l'algorithme glouton

```
def rendu_monnaie_rec(P,X):
  if X==0:
    return 0
  else:
    mini = 1000
  for i in range(len(P)):
    if P[i]≤X:
      nb = 1 + rendu_monnaie_rec(P,X-P[i])
      if nb<mini:
        mini = nb
  return mini

pieces = (2,5,10,50,100)
```