Algorithmique et Programmation 2

3. Types abstraits classiques

Lionel.Moisan@parisdescartes.fr

Université Paris Descartes
http://www.mi.parisdescartes.fr/∽moisan/

Types abstraits classiques

Pseudo-code et types abstraits

Avantages du pseudo-code

Un **pseudo-code** permet de formuler un algorithme en langage naturel :

- il est indépendant de tout langage de programmation
- il n'est pas soumis à des règles syntaxiques strictes (conventions usuelles : $\leftarrow$ pour l'affectation, // pour un commentaire, indentation des structures de contrôle, etc.)
- il permet de décrire la structure d'ensemble d'un algorithme sans aller immédiatement dans le détail des différentes étapes
- il permet de séparer l'écriture d'un programme en deux étapes :
  1. l'écriture de l'algorithme en pseudo-code, sans contrainte de syntaxe
  2. la traduction de ce pseudo-code dans le langage choisi

pseudo-code

fonction factorielle(n)
- $P \leftarrow 1$ // initialisation
- pour $k = 2, 3, \ldots n$
  - $P \leftarrow P \times k$
- retourner P

implémentation Python

```
def factorielle(n):
    P = 1 # initialisation
    for k in range(2,n+1):
        P = P * k
    return P
```

Exemple de pseudo-code : pivot de Gauss

Algorithme du pivot de Gauss pour résoudre un système linéaire (MC1) :

```
fonction pivot_gauss(A,b)
    // résout le système linéaire A x = b
    // A = (a_ij)_{1≤i,j≤n} est une matrice (carrée n × n) inversible
    // b = (b_j)_{1≤j≤n} est un vecteur de taille n

    x ← b

    pour i = 1, 2, ... n // boucle sur les lignes de A
        // normalisation de la ligne i pour mettre le pivot à 1
        diviser x_i et la ligne i de A par a_ii // on suppose a_ii ≠ 0
        pour j = i + 1, i + 2, ... n // boucle sur les lignes sous le pivot
            // fait apparaître un zéro dans A en position (j, i)
            x_j ← x_j − a_ji x_i
            soustraire à la ligne j de A la ligne i multipliée par a_ji
    // on fait apparaître de même des zéros au-dessus de la diagonale de A

    ...

    retourner x
```

Dans ce pseudo-code (incomplet), on ne se préoccupe pas de la syntaxe de manipulation des matrices, qui dépendra du langage utilisé

Exemple de pseudo-code : somme de nombres premiers

fonction somme(n)

```
    // retourne la somme des n premiers nombres premiers
    S ← 0 // somme cumulée
    p ← 2 // nombre courant
    k ← 0 // nombre de termes sommés dans S

    tant que k < n
        si p est un nombre premier
            S ← S + p
            k ← k + 1
        p ← p + 1
    retourner S
```

Dans cet algorithme, on ne détaille pas l'étape permettant de déterminer si p est premier ou non.

On ne peut donc pas l'implémenter directement dans un langage qui ne dispose pas nativement d'un test de primalité (il faudra implémenter d'abord cette fonction auxilliaire), mais il reste parfaitement compréhensible.

Exemple de pseudo-code : borne supérieure d'un polynôme

Détermination du maximum d'un polynôme sur $\mathbb{R}$:

```
fonction max_polynôme(P)
    // détermine la valeur maximale sur ℝ du polynôme P
    si P est un polynôme constant
        retourner P(0)
    si le degré de P est impair
        retourner +∞
    si le coefficient dominant de P est positif
        retourner +∞
    // P est de degré pair, de coefficient dominant négatif, donc majoré
    Q ← P′ // calcul de la dérivée de P
    m → −∞
    pour toute racine x de Q
        si P(x) > m
            m ← P(x)
    retourner m
```

Là encore, plusieurs opérations non détaillées !

Exemple du tri rapide (voir chapitre 2)

fonction tri(L) // retourne les éléments de la liste L dans l'ordre croissant

 si L est vide

  renvoyer L

 retirer un élément x de la liste L

 construire la liste L_1 de tous les éléments de L qui sont $< x$

 construire la liste L_2 de tous les éléments de L qui sont $\geq x$

 retourner la liste obtenue en concaténant tri(L_1), x, et tri(L_2)

Dans cet algorithme, on remarque :

- qu'on ne précise pas le choix de x $\rightarrow$ différentes variantes possibles
- que plusieurs étapes font référence à des opérations non élémentaires

Par ailleurs, on manipule ici une notion abstraite de liste (non liée à un langage), pour laquelle on peut effectuer les opérations suivantes :

- retirer un élement d'une liste
- parcourir l'ensemble des valeurs d'une liste
- concaténer plusieurs listes

$\rightarrow$ notion sous-jacente de **type abstrait** (ici, le type abstrait liste)

Notion de type abstrait

Définition

Un **type abstrait** est une définition mathématique d'un ensemble de données et des fonctions (et procédures) qui permettent de les manipuler.

En un sens, un type abstrait est à un type Python ce qu'un pseudo-code est à un code Python.

Un type abstrait est un type de données spécifié indépendamment d'un langage de programmation. Il sert à établir un cahier des charges des opérations que l'on veut pouvoir réaliser sur les données.

Ceci est utile car certains types de données ont une portée universelle (comme les nombres, les polynômes, les matrices en mathématiques).

L'implémentation d'un type abstrait consiste en l'écriture de fonctions permettant de remplir le cahier des charges. Un élément-clé est alors la complexité de ces opérations.

Types abstraits natifs en Python

Le type abstrait ensemble

Définition

Le type abstrait **ensemble** permet de représenter une collection finie d'objets (appelés éléments), non ordonnée et sans répétition.

Les fonctions de base associées sont :

- créer un ensemble vide :
  ensemble_vide() : $\rightarrow$ ensemble
- tester si un ensemble est vide :
  est_vide() : ensemble $\rightarrow$ booléen
- ajouter un élémént à un ensemble :
  ajouter_élément() : élément $\times$ ensemble (modifié) $\rightarrow$
- extraire un élément d'un ensemble (si l'ensemble est non vide)
  extraire_élément() : ensemble (modifié) $\rightarrow$ élément
- tester si un élément appartient à un ensemble :
  appartient_à() : élément $\times$ ensemble $\rightarrow$ booléen

Pour la fonction **extraire_élément()**, on note la présence d'une pré-condition, c'est-à-dire d'une condition sur les arguments de la fonction devant être vérifiée pour que l'appel de la fonction soit valide.

Types abstraits et complexité algorithmique

Le type abstrait ensemble est particulièrement utile :

- lorsqu'on veut éviter les doublons : si $x \in A$, ajouter_élément(x,A) n'a aucun effet sur A
- lorsqu'on veut tester efficacement la présence d'un élément dans un ensemble de grande taille : l'utilisation d'une fonction de hachage permet d'implémenter le calcul du booléen "$x \in A$" avec une complexité moyenne de $O(1)$ (temps constant indépendant de A), contre $O(n)$ pour la recherche d'un élément dans une liste de taille n.

De manière générale, même si les types abstraits ont un intérêt propre pour décrire des agencements de données et des algorithmes, la spécification de la complexité attendue des fonctions de base est indissociable de leur compréhension fine, puisqu'elle conditionne la complexité des algorithmes qui les utilisent.

Le type abstrait ensemble

Une implémentation riche du type ensemble ajoute en général les fonctions suivantes :

- **cardinal()** : ensemble $\rightarrow$ entier naturel
- **union()** : ensemble $\times$ ensemble $\rightarrow$ ensemble
- **intersection()** : ensemble $\times$ ensemble $\rightarrow$ ensemble
- **différence()** : ensemble $\times$ ensemble $\rightarrow$ ensemble
- **inclusion()** : ensemble $\times$ ensemble $\rightarrow$ booléen
- un **itérateur** permettant de décrire tous les éléments d'un ensemble, qui permet d'écrire, si E est un ensemble :

  pour tout x dans E

  ...
  <bloc d'instructions>
  ...

Exemple d'algorithme utilisant le type abstrait ensemble

Quiz : que calcule la fonction suivante ?

```
fonction f(A,B)
  // prend en entrée deux ensembles A et B
  C ← ensemble_vide()
  pour tout x dans B
    si appartient_à(x,A)
      ajouter_élément(x,C)
  retourner C
```

Réponse : f(A,B) retourne l'intersection de A et B

Exemple d'algorithme utilisant le type abstrait ensemble

Quiz : que calcule la fonction (récursive) suivante ?

```
fonction g(A)
  // prend en entrée un ensemble A et renvoie un ensemble
  B ← ensemble_vide()
  si est_vide(A)
      ajouter_élément(ensemble_vide(),B)
      retourner B
  x ← extraire_élément(A)
  E ← g(A)
  pour C dans E
      ajouter_élément(C,B)
      D ← C
      ajouter_élément(x,D)
      ajouter_élément(D,B)
  retourner B
```

Réponse : g(A) retourne l'ensemble des sous-ensembles de A

Exemple d'algorithme utilisant le type abstrait ensemble

Pseudo-code commenté :

```
fonction g(A)
  // prend en entrée un ensemble A
  // et renvoie l'ensemble de ses sous-ensembles
  B ← ensemble_vide() // initialisation du résultat : B = ∅
  si est_vide(A) // si A = ∅
    ajouter_élément(ensemble_vide(),B)
    retourner B // on retourne {∅}
  x ← extraire_élément(A) // on extrait x de A
  E ← g(A) // on calcule les sous-ensembles de A' = A\{x}
  pour C dans E // pour tout sous-ensemble C de A'
    ajouter_élément(C,B) // ajouter C au résultat B
    D ← C
    ajouter_élément(x,D)
    ajouter_élément(D,B) // ajouter C ∪ {x} au résultat B
  retourner B
```

Exemple d'algorithme utilisant le type abstrait ensemble

En pratique, on utilise souvent une écriture moins formelle :

```
fonction g(A)
    // prend en entrée un ensemble A
    // et renvoie l'ensemble de ses sous-ensembles
    B ← ensemble vide // initialisation du résultat : B = ∅
    si A est vide
        ajouter l'ensemble vide à B
        retourner B // on retourne {∅}
    extraire x de A
    E ← g(A) // on calcule les sous-ensembles de A' = A\{x}
    pour C dans E // pour tout sous-ensemble C de A'
        ajouter C à B
        D ← C
        ajouter x à D
        ajouter D à B // ajouter C ∪ {x} au résultat B
    retourner B
```

Voir la fonction génératrice `sous_ensembles()` définie au chapitre 2

Le type ensemble en Python

En Python, le type abstrait ensemble est natif, car déjà implémenté dans deux types de données : le type `set` (mutable) et le type `frozenset` (immutable). Le tableau ci-dessous donne l'équivalence entre les fonctions du type abstrait ensemble et l'implémentation Python.

fonction du type abstrait	type `set` Python	type `frozenset` Python
ensemble_vide()	`set()`	`frozenset()`
est_vide(E)	`len(E)==0`	`len(E)==0`
ajouter_élément(E,x)	`E.add(x)`	`E = E|{x}`
extraire_élément(E)	`E.pop()`	—
appartient_à(x,E)	`x in E`	`x in E`
cardinal(E)	`len(E)`	`len(E)`
union(E,F)	`E | F`	`E | F`
intersection(E,F)	`E & F`	`E & F`
différence(E,F)	`E - F`	`E - F`
inclusion(E,F)	`E <= F`	`E <= F`
itérateur sur E	`for x in E:`	`for x in E:`

Le type abstrait dictionnaire

Définition

Le type abstrait dictionnaire (aussi appelé tableau associatif) permet de représenter une collection finie d'objets (appelés clés), non ordonnée et sans répétition, avec une valeur associée à chaque clé.

Les fonctions de base associées sont :

- créer un dictionnaire vide :
  dictionnaire_vide() : $\rightarrow$ dictionnaire
- tester si un dictionnaire est vide :
  est_vide() : dictionnaire $\rightarrow$ booléen
- ajouter un couple (clé,valeur), ou changer la valeur d'une clé existante :
  spécifier_valeur() : clé $\times$ valeur $\times$ dictionnaire (modifié) $\rightarrow$
- tester si une clé est valide :
  clé_valide() : clé $\times$ dictionnaire $\rightarrow$ booléen
- retourner la valeur associée à une clé (seulement si la clé est valide)
  valeur() : clé $\times$ dictionnaire $\rightarrow$ valeur
- supprimer une clé (seulement si la clé est valide)
  supprimer_clé() : clé $\times$ dictionnaire (modifié) $\rightarrow$
- un **itérateur** sur toutes les clés d'un dictionnaire

Le type dictionnaire en Python

En Python, le type abstrait dictionnaire est natif, implémenté dans le type `dict`. Le tableau ci-dessous donne l'équivalence entre les fonctions du type abstrait dictionnaire et l'implémentation Python.

fonction du type abstrait	type `dict` Python
dictionnaire_vide()	`dict()`
est_vide(D)	`len(D) == 0`
spécifier_valeur(k, v, D)	`D[k] = v`
clé_valide(k, D)	`k in D`
valeur(k, D)	`D[k]`
supprimer_clé(k, D)	`del D[k]`
itérateur sur les clés de D	`for k in D:`

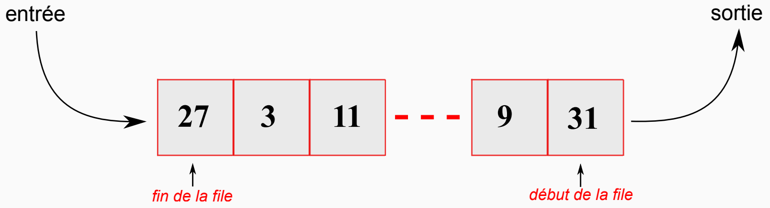
Files et piles

Les types abstraits file et pile

Définition générale

Les files et les piles sont des collections ordonnées d'éléments, auxquels on accède de manière limitée.

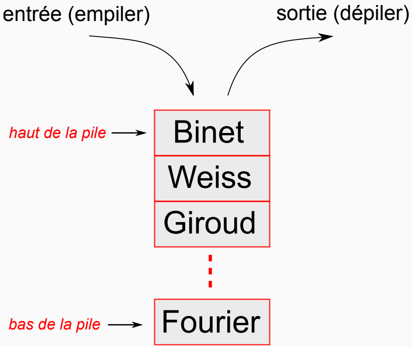
Dans une $\boxed{\text{file}}$, on peut extraire l'élément au début de la file (si la file est non vide) et on peut ajouter un élément à la fin de la file. C'est le principe d'une file d'attente à un guichet par exemple. On désigne ce principe d'accès aux éléments par l'acronyme $\boxed{\text{FIFO}}$ (*First In, First Out*).



Il existe des variantes de ce type abstrait (par exemple, les files de priorité), mais le principe de base reste le même.

Les types abstraits file et pile

Dans une $\boxed{\text{pile}}$, on peut extraire (= dépiler) l'élément au sommet de la pile (si la pile est non vide), et on peut ajouter (empiler) un nouvel élément au sommet de la pile. C'est le principe d'une pile d'assiettes que l'on manipulerait une par une. On désigne ce principe d'accès aux éléments par l'acronyme $\boxed{\text{LIFO}}$ (*Last In, First Out*).



Les types abstraits file et pile

Le type abstrait file est utile pour modéliser, par exemple :

- l'accès séquentiel à des ressources (accès réseau, file d'impression)
- la gestion de clients, de requètes, etc. (files d'attente)
- la modélisation d'un système régulier de transport (métro, bus, etc.)
- tout ce qui relève du principe "premier arrivé, premier servi"

Le type abstrait pile est utile pour modéliser, par exemple :

- l'historique des modifications dans un traitement de texte ou autre (commande *Annuler modification*)
- l'historique de navigation dans un navigateur internet (commande *Reculer d'une page*)
- la récursivité

Le type abstrait file

Définition (fonctions de base pour les files)

Les fonctions de base associées au type file sont :

- créer une file vide :
  file_vide() : $\rightarrow$ file
- tester si une file est vide :
  est_vide() : file $\rightarrow$ booléen
- ajouter un élément à la fin d'une file :
  ajouter_élément() : élément $\times$ file (modifiée)$\rightarrow$
- extraire le premier élément d'une file (si la file est non vide) :
  extraire_élément() : file (modifiée) $\rightarrow$ élément
- lire le premier élément de la file (si la file est non vide) :
  premier_élément() : file $\rightarrow$ élément

Le type abstrait file

Quiz : que vaut la file F après l'exécution du pseudo-code suivant ?

```
F ← file_vide()
ajouter_élément(5,F)
ajouter_élément(3,F)
tant que non(est_vide(F)) et premier_élément(F)>2
    x ← extraire_élément(F)
    ajouter_élément(x-1,F)
    ajouter_élément(x-3,F)
```

Réponse :

avant la boucle :	F =					5
puis	F =				3	5
après le premier tour de boucle :	F =			2	4	3
après le deuxième tour de boucle :	F =		0	2	2	4
après le troisième tour de boucle :	F =	1	3	0	2	2

Implémentation Python du type abstrait file

Le type abstrait file peut être implémenté à partir du type `list`. On choisit ci-dessous d'ajouter les éléments à gauche de la liste (donc à l'indice 0, en décalant les autres), et de les extraire à droite (dernier indice valide).

```
def file_vide(): # -> file
    return []
def est_vide(F): # file -> booléen
    return len(F)==0
def ajouter_élément(x,F): # élément x file -> file
   return [x]+F
def extraire_élément(F): # file (modifiée) -> élément
    return F.pop()
def premier_élément(F): # file -> élément
    return F[-1]
```

Cette implémentation est correcte mais pas complètement satisfaisante car l'ajout d'un élément nécessite de recopier la liste existante
→ peu efficace si la file est très grande

Implémentation Python du type abstrait file

On peut également implémenter le type file dans l'autre sens, avec l'ajout des éléments à droite (en fin de liste) et l'extraction à gauche (indice 0).

```
def file_vide(): # -> file
    return []
def est_vide(F): # file -> booléen
    return len(F)==0
def ajouter_élément(x,F): # élément x file (modifiée) ->
    F.append(x)
def extraire_élément(F): # file  -> élément x file
    return F[0],F[1:]
def premier_élément(F): # file -> élément
    return F[0]
```

Cette implémentation est cette fois inefficace pour l'extraction (pour une liste de taille n, complexité $O(n)$ au lieu de $O(1)$).

Une meilleure implémentation peut être obtenue grâce au type `deque` (*double-ended queue*) du module `collections` (hors programme).

Le type abstrait pile

Définition (fonctions de base pour les piles)

Les fonctions de base associées au type pile sont :

- créer une pile vide :
  pile_vide() : $\rightarrow$ pile
- tester si une pile est vide :
  est_vide() : pile $\rightarrow$ booléen
- empiler un élément au sommet de la pile :
  empiler() : élément $\times$ pile (modifiée) $\rightarrow$
- dépiler l'élément en haut de la pile (si la pile est non vide) :
  dépiler() : pile (modifiée) $\rightarrow$ élément
- lire l'élément au sommet de la pile (si la pile est non vide) :
  sommet() : pile $\rightarrow$ élément

Aux noms près, les fonctions de base sont similaires à celles du type file, mais elles diffèrent dans leur effet en raison de la différence FIFO / LIFO.

Quiz : que vaut la pile P après l'exécution du pseudo-code suivant ?

```
P ← pile_vide()
empiler(5,P)
empiler(3,P)
tant que non(est_vide(P)) et sommet(P)>2
    x ← dépiler(P)
    empiler(x-1,P)
    empiler(x-3,P)
```

Réponse :

avant la boucle

P =

5

puis P =

3
5

après le premier tour de boucle

P =

0
2
5

Implémentation Python du type abstrait pile

Le type abstrait pile peut être implémenté à partir du type `list`.

```
def pile_vide(): # -> pile
    return []
def est_vide(P): # pile -> booléen
    return len(P)==0
def empiler(P,x): # pile (modifiée) x élément ->
    P.append(x)
def dépiler(P): # pile (modifiée) -> élément
    return P.pop()
def sommet(P): # pile -> élément
    return P[-1]
```

Cette implémentation est efficace (et conforme à ce qui est attendu) car toutes ces fonctions s'exécutent en **temps constant** (la complexité est $O(1)$, c'est-à-dire majorée par une constante indépendante de la taille de la pile)

Piles et récursivité

À peu près tous les langages de programmation utilisent des piles (*stack* en anglais), en particulier pour gérer les appels de fonctions et la récursivité.

Par exemple, si une fonction `f` a été définie par `def f(a,b):` l'instruction $x = f(1,3)$ se traduit pour le système par :

- empiler 1, puis 3
- appeler la fonction f, qui va :
  - utiliser les deux emplacements mémoire au sommet de la pile comme variables locales pour ses deux arguments ($a = 1$ et $b = 3$)
  - ...faire des calculs...
  - à l'instruction `return`, libérer (dépiler) les deux emplacements utilisés par a et b au sommet de la pile
  - empiler la valeur de retour
  - rendre la main au programme appelant
- dépiler la valeur de retour de f dans x

Piles et récursivité

Examinons la pile du système lors de l'appel `pgcd(9,6)`, où `pgcd()` est l'implémentation récursive du calcul du pgcd par l'algorithme d'Euclide (voir chapitre 2) :

```
def pgcd(a,b):
    return a if b==0 else pgcd(b,a%b)
```

étape →	0	1	2	3	4	5	6
pgcd(9,6)		appel	—	—	—	—	return
pgcd(6,3)			appel	—	—	return	
pgcd(3,0)				appel	return		
pile ⟶		6 9	3 6 6 9	0 3 3 6 6 9	3 3 6 6 9	3 6 9	3

Exemple d'utilisation d'une pile : vérifier le parenthésage

On souhaite analyser une chaîne de caractères et déterminer si elle est correctement parenthésée avec les symboles ( ) [ ]

Exemples :

- `[(x+1)/z-f(y)]/2` est correctement parenthésée
- `3((1+x)` n'est pas bien parenthésée
- `(1+f][3x-2)` n'est pas bien parenthésée

Autrement dit, la chaîne de caractères est bien parenthésée si lorsqu'on tombe sur un symbole de fermeture) ou], le dernier symbole d'ouverture *non déjà refermé* correspond au symbole d'ouverture équivalent (ou [.

On peut faire cette vérification à l'aide d'une pile :

- on balaie les caractères un à un, dans l'ordre
- lorsqu'on tombe sur un symbole ouvrant, on l'empile
- lorsqu'on tombe sur un symbole fermant, on vérifie que la pile est non vide, puis que l'élément que l'on dépile est bien le symbole ouvrant correspondant
- à la fin de la chaîne, la pile doit être vide

Exemple d'utilisation d'une pile : vérifier le parenthésage

L'algorithme de vérification peut se décrire avec le pseudo-code suivant :

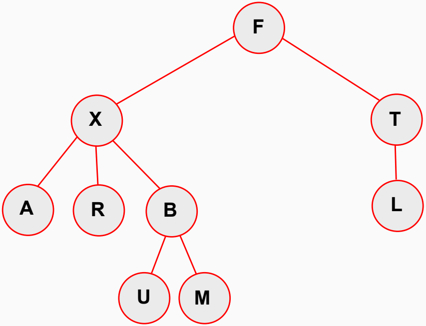
```
fonction bien_parenthésée(s)
    // teste si la chaîne de caractères s est bien parenthésée avec ( ) et [ ]
    // signature : chaîne de caractères → booléen
    P ← pile_vide() // initialisation
    pour tout caractère c de s
        si c = '(' ou c = '[' // symbole ouvrant
            empiler(c,P)
        sinon si c = ')'
            si pile_vide(P), retourner FAUX // manque (
            si dépiler(P)≠ '(', retourner FAUX // incohérence
        sinon si c = ']'
            si pile_vide(P), retourner FAUX // manque [
            si dépiler(P)≠ '[', retourner FAUX // incohérence
    si non(est_vide(P)), retourner FAUX // symboles ouvrants non refermés
    retourner VRAI // tout a été vérifié
```

Arbres

Le type abstrait arbre

Première définition

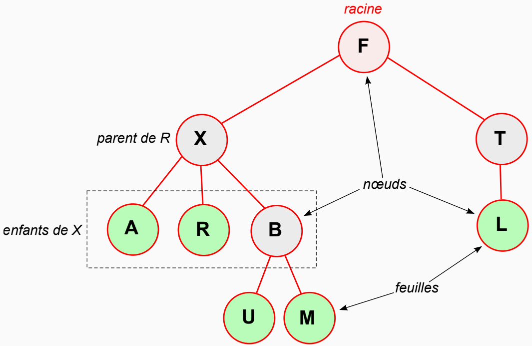
Un **arbre** est une structure de données hiérarchique constituée de nœuds. A l'exception d'un nœud particulier appelé racine de l'arbre, chaque nœud a un unique parent (qui est un autre nœud de l'arbre). Réciproquement, un nœud peut avoir 0, un ou plusieurs enfants dont il est le parent. Un nœud sans enfant est appelé feuille.



Le type abstrait arbre

Première définition

Un **arbre** est une structure de données hiérarchique constituée de nœuds. A l'exception d'un nœud particulier appelé racine de l'arbre, chaque nœud a un unique parent (qui est un autre nœud de l'arbre). Réciproquement, un nœud peut avoir 0, un ou plusieurs enfants dont il est le parent. Un nœud sans enfant est appelé feuille.



Le type abstrait arbre

On appelle branche le lien entre un nœud et son parent.

Dans un arbre, les nœuds (et, plus rarement, les branches) contiennent généralement une information (on parle de l'étiquette, du label, ou plus simplement du nom du nœud).

La profondeur d'un nœud est le nombre de parents à remonter pour atteindre la racine (qui est de profondeur 0).

La hauteur d'un arbre est la profondeur maximale de ses nœuds (donc de ses feuilles en fait).

La taille d'un arbre est son nombre de nœuds.

Un arbre peut être ordonné (si les enfants sont numérotés) ou non ordonné (on parle alors de l'ensemble des enfants)

Remarque : il y a une analogie avec les arbres biologiques, mais en informatique on les représente dans l'autre sens, avec la racine en haut et les feuilles en bas !

Le type abstrait arbre

Deuxième définition (récursive)

Un objet A est un **arbre** si :

- soit A est l'arbre vide ;
- soit A est un arbre constitué d'un nœud appelé racine de A, dont les enfants sont eux-mêmes des arbres.
  Le nombre d'enfants peut être nul.

Le type abstrait arbre se rencontre dans de nombreuses situations :

- les objets graphiques d'une application (fenêtres, menus, etc.)
- les représentations hiérarchiques
- la structure d'un système de fichier
- la représentation d'une formule algébrique
- l'établissement d'un diagnostic (arbre de décision)
- etc.

Les arbres se prètent particulièrement bien aux algorithmes récursifs

Le type abstrait arbre

Définition (fonctions de base pour les arbres)

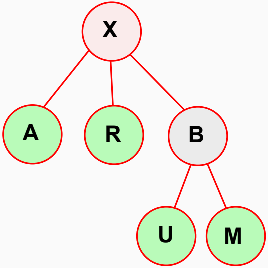
Les fonctions de base associées au type arbre (ordonné) sont :

- créer un nœud à partir d'une étiquette et d'une liste de nœuds enfants (si la liste est vide, on crée une feuille) :
  créer_noeud() : étiquette $\times$ liste de nœuds $\rightarrow$ nœud
- lire l'étiquette d'un nœud :
  étiquette() : nœud $\rightarrow$ étiquette
- lire la liste des enfants d'un nœud :
  enfants() : nœud $\rightarrow$ liste de nœuds
- créer un arbre dont la racine est donnée par un nœud :
  créer_arbre() : nœud $\rightarrow$ arbre
- lire la racine d'un arbre :
  racine() : arbre $\rightarrow$ nœud

Le type abstrait arbre

Quiz : que vaut l'arbre a après l'exécution du pseudo-code suivant ?

U ← créer_noeud('U',[ ])
M ← créer_noeud('M',[ ])
B ← créer_noeud('B',[U,M])
A ← créer_noeud('A',[ ])
R ← créer_noeud('R',[ ])
X ← créer_noeud('X',[A,R,B])
a ← créer_arbre(X)



Implémentation Python du type abstrait arbre

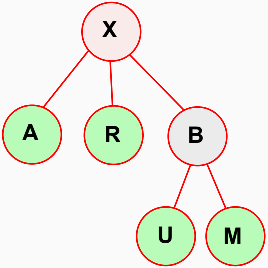
On peut implémenter simplement le type abstrait arbre (ordonné) à l'aide de listes (type `list`).

Chaque nœud est représenté par une liste dont le premier élément est l'étiquette, et les éléments suivants les enfants.

```python
def creer_noeud(e,L): # étiquette x liste de noeuds -> noeud
    return [e]+L

def etiquette(N): # noeud -> étiquette
    return N[0]

def enfants(N): # noeud -> liste de noeuds
    return N[1:]

def creer_arbre(N): # noeud -> arbre
    return N # ici, on identifie un arbre avec sa racine

def racine(A): # arbre -> noeud
    return A # même remarque
```

Le type abstrait arbre

Quiz : selon l'implémentation précédente, qu'elle liste `L` coderait l'arbre ci-dessous ?



Réponse :

```
L = ['X',['A'],['R'],['B',['U'],['M']]]
```

Le type abstrait arbre : autres fonctions de base

Dans certains algorithmes, il est utile de disposer de fonctions pour pouvoir "remonter" dans l'arbre :

- tester si un nœud est la racine (= n'a pas de parent) :
  est_racine() : nœud $\rightarrow$ booléen
- lire le parent d'un nœud (si ce nœud n'est pas la racine) :
  parent() : nœud $\rightarrow$ nœud

Pour implémenter efficacement en Python ces fonctions de base, on peut modifier la représentation précédente en ajoutant le parent dans la structure liste, ou bien utiliser une structure de données "objet" spécifique.

D'autres fonctions de base peuvent également être ajoutées (supression de feuilles ou de nœuds, création et test de l'arbre vide, modification d'une étiquette, déplacement d'un nœud, etc.)

Parcours d'un arbre

Pour visiter tous les nœuds d'un arbre, on peut utiliser un algorithme récursif très simple appelé **parcours en profondeur** :

- on part de la racine
- le parcours d'un nœud consiste à parcourir successivement ses enfants

Le parcours d'un arbre A s'écrit ainsi simplement explore(racine(A)), où la fonction explore() est définie par le pseudo-code suivant :

```
fonction explore(N)
    // explore le sous-arbre défini par le nœud N
    traiter le nœud N (selon l'objectif du parcours)
    pour tout enfant E de N
        explore(E)
```

Il existe d'autres méthodes de parcours d'un arbre, notamment le parcours en largeur que nous verrons plus tard.

Exemples de fonctions simples sur les arbres

Grâce au parcours en profondeur que nous venons de voir, nous pouvons implémenter très simplement les fonctions suivantes :

- Calcul de la taille d'un arbre A avec taille(racine(A)) :

```
fonction taille(N)
    // retourne la taille (nombre de nœuds) du sous-arbre de nœud N
    T ← 1 // compte le nœud N
    pour tout enfant E de N
        T ← T + taille(E)
    retourner T
```

- Calcul de la hauteur d'un arbre A avec hauteur(racine(A)) :

```
fonction hauteur(N)
    // retourne la hauteur (profondeur maximale) du sous-arbre de nœud N
    T ← 0 // initialisation
    pour tout enfant E de N
        T ← max( T , hauteur(E) )
    T ← T + 1 // ajoute le niveau du nœud N
    retourner T
```

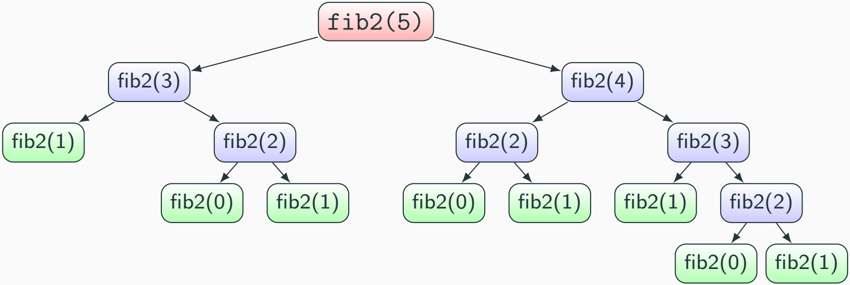
Arbre des appels récursifs

Lors de l'appel d'une fonction récursive, on peut représenter les appels de la fonction sous la forme d'un arbre (si un appel A provoque l'appel B, alors A est parent de B).

Nous avons vu au chapitre 2 la fonction doublement récursive `fib2()` :

```
def fib2(n):
    return n if n<=1 else fib2(n-2)+fib2(n-1)
```

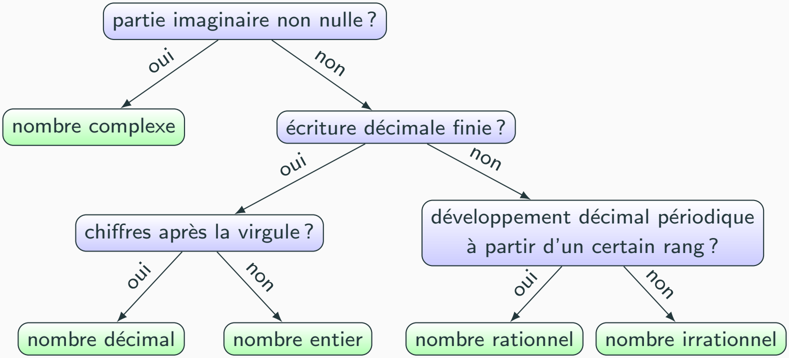
L'appel `fib2(5)` produit l'arbre des appels récursifs suivant :



Arbre de décision

Un arbre de décision est un arbre dont les nœuds internes (c'est-à-dire les nœuds qui ne sont pas des feuilles) sont des tests et les feuilles des éléments de classification ou de décision.

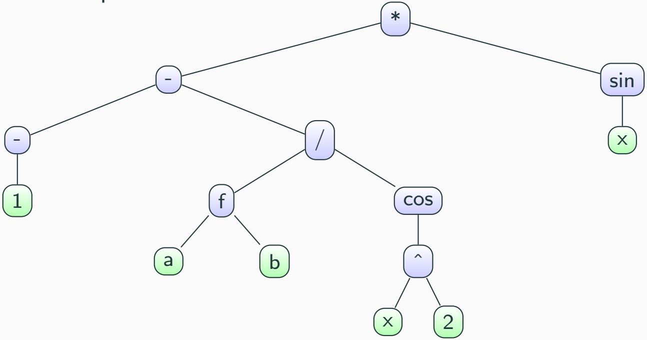
Exemple : classification d'un nombre



Représentation d'une expression algébrique

Les expressions algébriques se manipulent plus facilement par des algorithmes lorsqu'elles sont représentées par des arbres. Les nœuds sont des fonctions ou des opérateurs, les feuilles des variables ou des nombres.

Par exemple, l'expression algébrique `(-1-f(a,b)/cos(x^2))*sin(x)` sera représentée par l'arbre



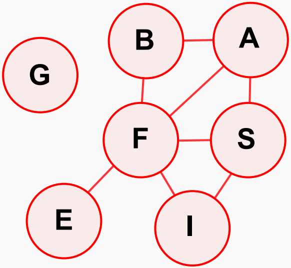
L'opérateur '-' est ici présent à la fois sous sa forme unaire (opposé) et binaire (soustraction).

Graphes

Définition d'un graphe

Définition

Un **graphe** (simple non orienté) est constitué d'un ensemble de sommets, et d'un ensemble d'arêtes qui relient entre eux certains couples de sommets. On peut le représenter mathématiquement par un couple $(\mathcal{S}, \mathcal{A})$, où S est l'ensemble des sommets et $\mathcal{A} \subset \big\{\{x, y\}, (x, y) \in \mathcal{S}^2, x \neq y\big\}$ l'ensemble des arêtes.



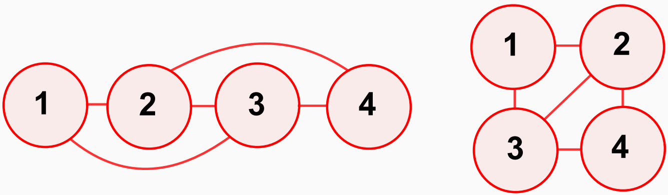
$\mathcal{S} = \{G, B, A, F, S, E, I\}$

$\mathcal{A} = \big\{\{B, A\}, \{B, F\}, \{A, F\}, \{A, S\}, \{F, S\}, \{F, E\}, \{F, I\}, \{S, I\}\big\}$

Dessin d'un graphe

Attention de ne pas confondre le graphe avec le dessin du graphe !

Par exemple, les deux graphes ci-dessous sont les mêmes !



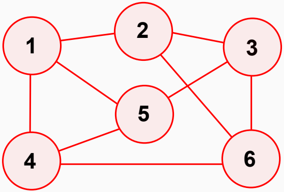
Vocabulaire des graphes

L'ordre d'un graphe est son nombre de sommets.

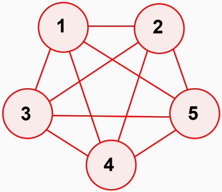
Dans un graphe, on appelle degré d'un sommet le nombre d'arêtes qui partent de ce sommet. Un graphe dont tous les sommets ont le même degré est dit régulier.

Dans un graphe non orienté, deux sommets reliés par une arête sont dits adjacents ou voisins.

Un graphe est complet si toutes les arêtes possibles sont présentes (autrement dit, tout sommet est voisin de tous les autres sommets).



Graphe régulier (sommets de degré 3)



Graphe complet d'ordre 5

Le type abstrait graphe

Définition (fonctions de base pour les graphes)

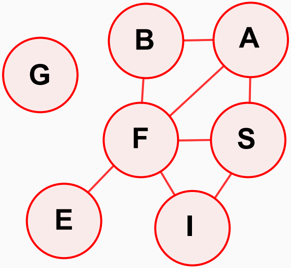
Les fonctions de base associées au type graphe (simple non orienté) sont :

- créer un graphe vide :
  graphe_vide() : $\times$ graphe
- tester si un graphe est vide :
  est_vide() : graphe $\rightarrow$ booléen
- ajouter un sommet à un graphe, en précisant ses voisins :
  (les voisins qui n'existent pas ne sont pas créés)
  ajouter_sommet() :
  graphe (modifié) $\times$ sommet $\times$ ensembles de sommets $\rightarrow$
- obtenir l'ensemble des voisins d'un sommet :
  voisins() : graphe $\times$ sommet $\rightarrow$ ensemble de sommets
- obtenir la liste des sommets d'un graphe :
  sommets() : graphe $\rightarrow$ liste de sommets

Le type abstrait graphe

Quiz : que vaut le graphe g après l'exécution du pseudo-code suivant ?

```
g ← graphe_vide()
ajouter_sommet(g,G,{})
ajouter_sommet(g,B,{})
ajouter_sommet(g,A,{B})
ajouter_sommet(g,F,{B,A})
ajouter_sommet(g,S,{F,A})
ajouter_sommet(g,E,{F})
ajouter_sommet(g,I,{F,S})
```



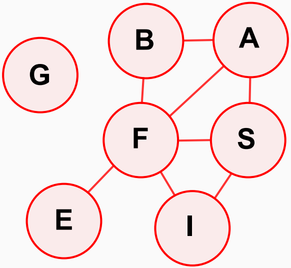
Implémentation Python du type abstrait graphe

On peut implémenter un graphe (simple non orienté) au moyen d'un dictionnaire :

- les sommets sont les clés
- la valeur associée à un sommet *s* est l'ensemble des voisins de *s*

```
def graphe_vide(): # -> graphe
    return dict()

def ajouter_sommet(G,s,A):
    # graphe (modifié) x sommet x ensemble de sommets ->
    G[s] = A # ajout du sommet s et des arêtes partant de s
    for t in A: # pour tout voisin de s
        if t in G:
            G[t].add(s) # ajout de l'arête de t vers s

def voisins(G,s): # graphe x sommet -> ensemble de sommets
    return G[s]

def sommets(G): # graphe -> liste de sommets
    return G.keys()
```

Implémentation Python du type abstrait graphe




```
>>> g = graphe_vide()
>>> ajouter_sommet(g,'G',set())
>>> ajouter_sommet(g,'B',set())
>>> ajouter_sommet(g,'A',{'B'})
>>> ajouter_sommet(g,'F',{'B','A'})
>>> ajouter_sommet(g,'S',{'F','A'})
>>> ajouter_sommet(g,'E',{'F'})
>>> ajouter_sommet(g,'I',{'F','S'})
>>> g
{'E':{'F'}, 'I':{'S','F'}, 'B':{'F','A'}, 'S':{'F','I','A'},
 'G':set(), 'F':{'S','E','I','B','A'}, 'A':{'S','F','B'}}
```

Implémentation Python du type abstrait graphe

On peut compléter les fonctions de base précédentes pour pouvoir modifier un graphe existant :

```
def supprimer_sommet(G,s): # graphe (modifié) x sommet ->
    for t in G[s]:
        G[t].remove(s) # on supprime les arêtes vers s
    del G[s] # on supprime le sommet s

def supprimer_arete(G,s,t):
    # graphe (modifié) x sommet x sommet ->
    G[s].remove(t)
    G[t].remove(s)

def ajouter_arete(G,s,t):
    # graphe (modifié) x sommet x sommet ->
    G[s].add(t)
    G[t].add(s)
```

L'utilisation des types Python `dict` et `set` permet d'obtenir, pour toutes les fonctions de base, une complexité moyenne constante $O(1)$ (donc indépendante du nombre de sommets et d'arêtes dans le graphe)

Autres implémentations du type abstrait graphe

On peut également implémenter le type abstrait graphe (simple non orienté) :

- avec des listes (un sommet est une liste composée de son nom et de ses voisins) ;
- par la matrice d'adjacence du graphe (voir définition plus loin) ;
- par un ensemble de sommets et un ensemble d'arêtes (implémentation à la lettre de la définition mathématique) ;
- etc.

En termes de complexité, ces méthodes sont moins efficaces que l'implémentation par dictionnaire. Elles peuvent néanmoins avoir un intérêt pour des graphes de taille modérée.

Pour l'implémentation par liste d'adjacence par exemple, l'ajout d'un sommet ou d'une arête, et la détermination des voisins d'un sommet, ont une complexité linéaire (et non constante) en le nombre de sommets.

Exemple d'algorithme

Vérifier qu'un graphe est complet, autrement dit que chaque sommet est voisin de tous les autres :

fonction est_complet(G)
  // retourne vrai si le graphe G est complet, faux sinon
  S $\leftarrow$ ensemble formé par les sommets de G
  pour tout élémént s de S
    si $\{s\} \cup$ voisins(G,s) $\neq$ S
      retourner faux
  retourner vrai

Implémentation Python :

```
def est_complet(G): # graphe -> booléen
    S = set(sommets(G))
    for s in S:
        if {s}|voisins(G,s) != S:
            return False
    return True
```

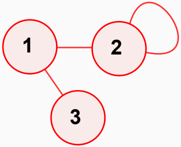
Les graphes en informatique

Le graphes sont extrêmement utiles en informatique, et on les utilise dans des contextes très différents :

- pour la modélisation de réseaux (internet, chemin de fer, lignes aériennes, téléphonie mobile, électricité, réseaux sociaux, etc.)
- en robotique, et plus généralement pour l'automatisation de tâches (les sommets sont des états de la machine, les arêtes des actions)
- pour la logistique (planification du transport de produits chimiques incompatibles)
- en modélisation biologique (ex : propagation d'une épidémie)
- en modélisation de phénomènes physiques (ex : percolation)
- en théorie des jeux (par exemple, on peut voir une partie d'échecs comme une promenade sur un graphe dont les sommets sont les configurations et les arêtes des coups valides)
- et beaucoup d'autres !

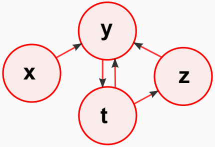
autres types de graphes

- un graphe est simple s'il ne comporte pas de boucles, c'est-à-dire d'arêtes dont les deux extrémités sont identiques

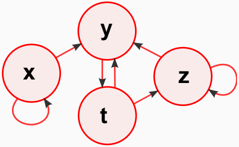


ce graphe n'est pas simple

- un graphe orienté (simple ou non) est un graphe dans lequel les arêtes sont dirigées (l'arête x→y va de x vers y). Il peut être représenté par un couple $(\mathcal{S}, \mathcal{A})$, avec $\mathcal{A} \subset \mathcal{S}^2$.



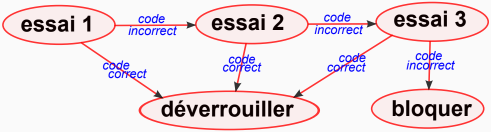
graphe orienté simple



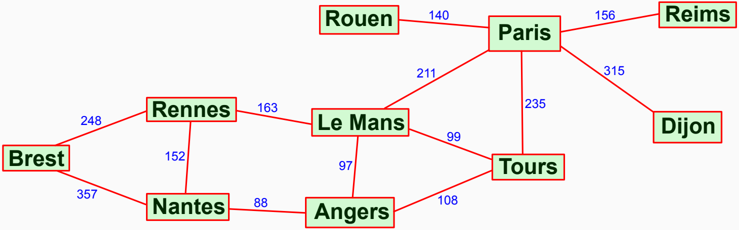
graphe orienté non simple

autres types de graphes

- Un graphe étiqueté (orienté ou non) est un graphe dont les arêtes sont munies d'étiquettes. Lorsque ces étiquettes sont des réels positifs, on a un graphe pondéré.



graphe étiqueté orienté (représentation d'un automate)



graphe pondéré non orienté (distances SNCF en km)

Chemins et cycles

Dans un graphe, un **chemin** (ou une chaîne) est une liste de sommets dans laquelle deux sommets consécutifs sont adjacents.

La longueur d'une chemin est le nombre d'arêtes qui la constitue (donc, le nombre de sommets moins un).

Une chemin est fermé si ses deux extrémités (son premier et son dernier élément) sont identiques.

Un cycle est un chemin fermé dont toutes les arêtes sont distinctes.

Définition

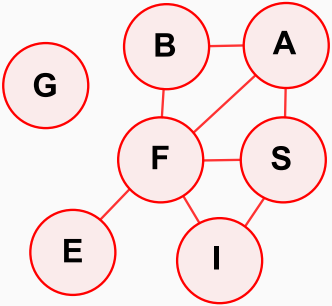
Un graphe est **connexe** si deux sommets quelconques sont connectés, c'est-à-dire, peuvent être reliés par un chemin dont ils sont les deux extrémités.

Théorème

Un graphe est connexe si et seulement si il existe un chemin passant par tous les sommets.

Exemples de graphe

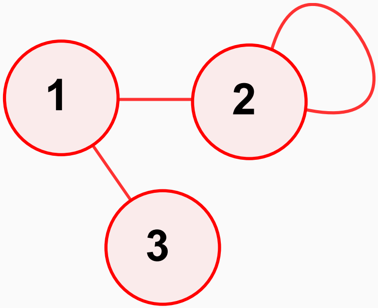
Quiz : remplir le tableau pour le graphe ci-dessous



simple	connexe	d^{o} min	d^{o} max	régulier	complet	ordre	cycles
oui	non	0	5	non	non	7	oui

Exemples de graphe

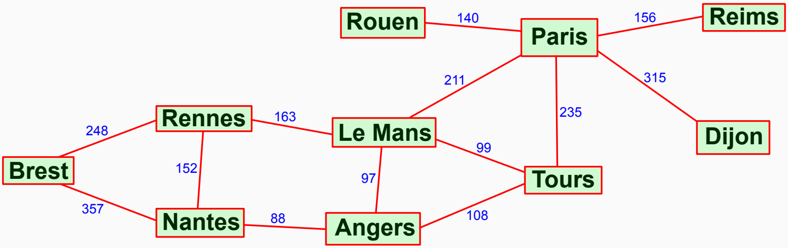
Quiz : remplir le tableau pour le graphe ci-dessous



simple	connexe	d° min	d° max	régulier	complet	ordre	cycles
non	oui	1	3	non	non	3	oui

Exemples de graphe

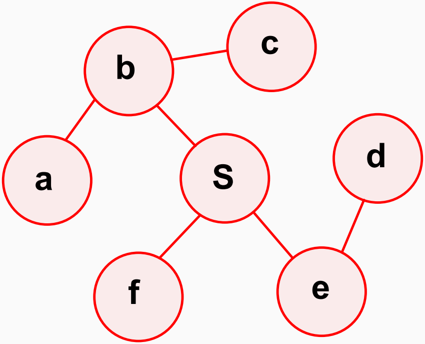
Quiz : remplir le tableau pour le graphe ci-dessous



simple	connexe	d° min	d° max	régulier	complet	ordre	cycles
oui	oui	1	5	non	non	10	oui

Exemples de graphe

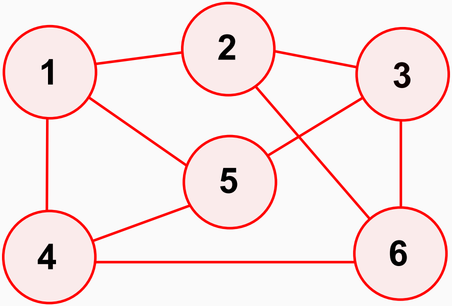
Quiz : remplir le tableau pour le graphe ci-dessous



simple	connexe	d° min	d° max	régulier	complet	ordre	cycles
oui	oui	1	3	non	non	7	non

Exemples de graphe

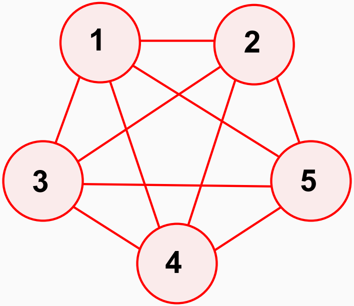
Quiz : remplir le tableau pour le graphe ci-dessous



simple	connexe	d° min	d° max	régulier	complet	ordre	cycles
oui	oui	3	3	oui	non	6	oui

Exemples de graphe

Quiz : remplir le tableau pour le graphe ci-dessous



simple	connexe	d° min	d° max	régulier	complet	ordre	cycles
oui	oui	4	4	oui	oui	5	oui

Matrice d'adjacence associée à un graphe

On peut représenter un graphe par sa **matrice d'adjacence**

On numérote les sommets de 1 à n

La matrice d'adjacence $A = (a_{ij})_{1 \leq i,j \leq n}$ du graphe est définie par

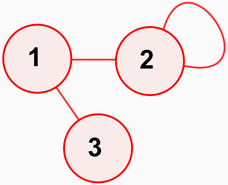
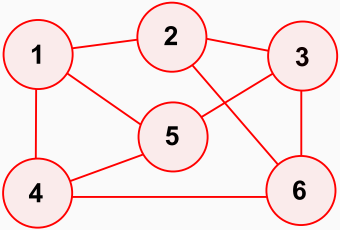
$$a_{ij} = \begin{cases} 1 \text{ s'il existe une arête allant du sommet } i \text{ au sommet } j, \\ 0 \text{ sinon.} \end{cases}$$

Elle définit complètement le graphe.

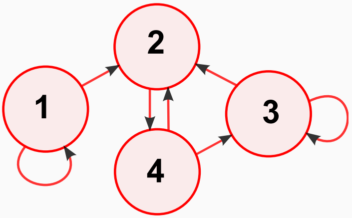
Si le graphe est non orienté, la matrice A est **symétrique** $(\forall i, j,\ a_{ji} = a_{ij})$

Si le graphe est simple, tous les coefficients diagonaux de A sont nuls $(\forall i,\ a_{ii} = 0)$

Matrice d'adjacence associée à un graphe

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

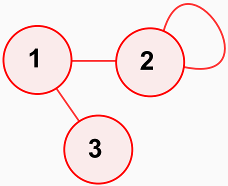
Matrice d’adjacence associée à un graphe

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

Théorème (dénombrement de chemins)

Si A est la matrice d'adjacence d'un graphe G, alors pour tout entier $k \in \mathbb{N}^*$, le coefficient (i,j) de la matrice A^k est le nombre de chemins distincts de longueur k allant du sommet i au sommet j dans G.

Exemple : G =



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

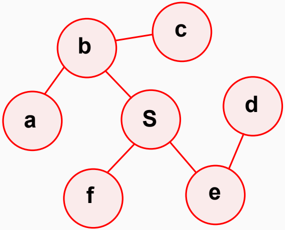
$$A^2 = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 1 \end{pmatrix} ; \quad A^3 = \begin{pmatrix} 1 & 3 & 2 \\ 3 & 3 & 1 \\ 2 & 1 & 0 \end{pmatrix} ; \quad A^4 = \begin{pmatrix} 5 & 4 & 1 \\ 4 & 6 & 3 \\ 1 & 3 & 2 \end{pmatrix} ; \quad \text{etc.}$$

$\rightarrow$ il y a 5 manières d'aller du sommet 1 à lui-même en 4 étapes (1-3-1-3-1, 1-3-1-2-1, 1-2-1-3-1, 1-2-1-2-1, 1-2-2-2-1)

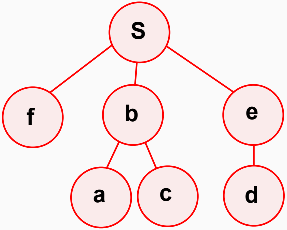
Graphes et arbres

Si l'on fait abstraction des différences de vocabulaire (nœud contre sommet, relation parent-enfant contre arête), un arbre peut-être vu comme un graphe dont un sommet est pointé (= distingué) comme la racine de l'arbre.

Réciproquement, tout graphe (non orienté) acyclique (c'est-à-dire ne comportant pas de cycle) dont on pointe un sommet S est équivalent à un arbre de racine S.

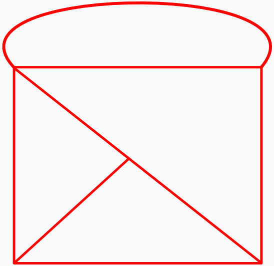


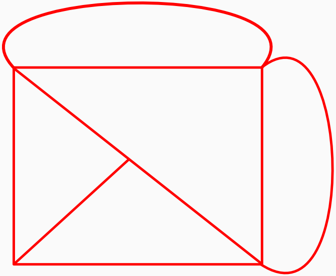
un graphe acyclique avec un sommet pointé (S)



l'arbre de racine S associé

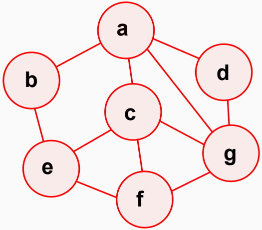
Question : peut-on tracer les figures suivantes sans lever le crayon ?
(et sans repasser sur le même trait plusieurs fois)





Définition

Dans un graphe simple non orienté, un chemin eulérien est une chemin qui contient exactement une fois chaque arête du graphe.

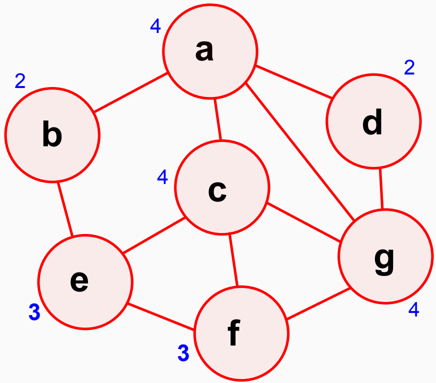


Ce graphe contient-il un chemin eulérien ?
$\rightarrow$ oui, par exemple la chemin e-b-a-c-g-d-a-g-f-c-e-f

Théorème d'Euler

Un graphe connexe admet une chemin eulérien si et seulement si il possède 0 ou 2 sommets de degré impair.

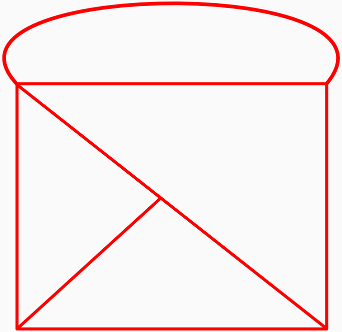
Chemins eulériens



En bleu, le degré de chaque sommet

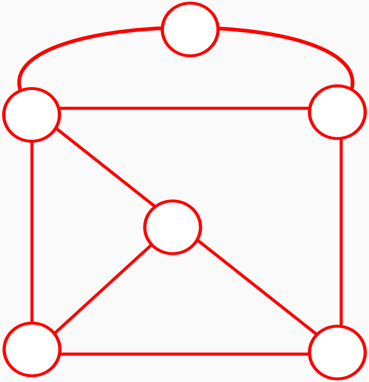
Il y a exactement 2 sommets de degré impair
$\rightarrow$ il existe une chemin eulérien

Tracé sans lever le crayon ?



Dessin initial

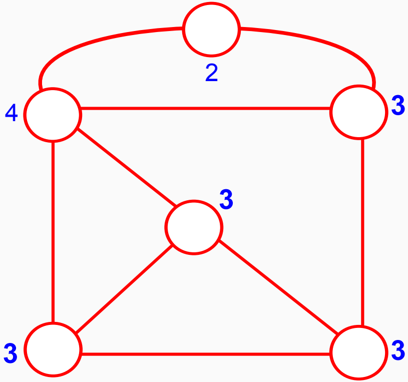
Tracé sans lever le crayon ?



Graphe associé

(on place des sommets aux intersections et pour distinguer les arêtes multiples)

Tracé sans lever le crayon ?



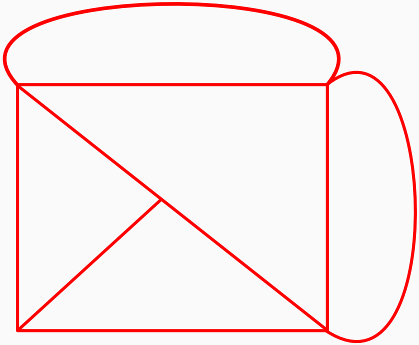
Avec le degré de chaque sommet indiqué en bleu

Il y a exactement 4 sommets de degré impair

→ pas de chemin eulérien

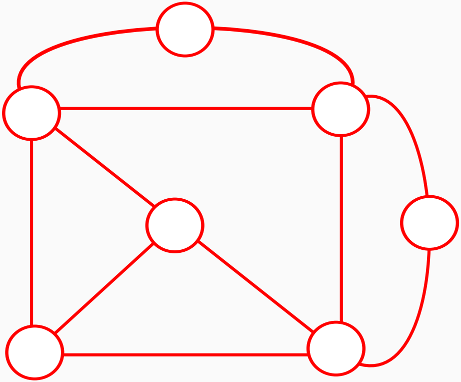
→ pas de tracé sans lever le crayon

Tracé sans lever le crayon ?



Dessin initial

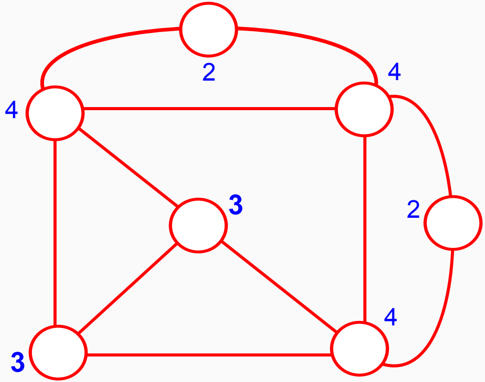
Tracé sans lever le crayon ?



Graphe associé

(on place des sommets aux intersections et pour distinguer les arêtes multiples)

Tracé sans lever le crayon ?



Avec le degré de chaque sommet indiqué en bleu

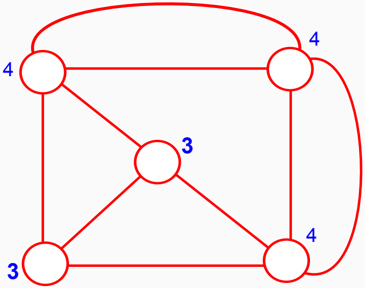
Il y a exactement 2 sommets de degré impair
$\rightarrow$ il existe un chemin eulérien
$\rightarrow$ on peut tracer la figure sans lever le crayon

Cas des multigraphes

Remarque : Le théorème d'Euler reste valable pour les multigraphes, c'est-à-dire des graphes pour lesquels on peut avoir plusieurs arêtes qui joignent deux mêmes sommets

Pour les deux exemples précédents, on a évité de considérer des multigraphes en introduisant artificiellement des sommets

Si l'on s'en tient au multigraphe obtenu en plaçant des sommets uniquement aux intersections, la conclusion est identique



Algorithme pour l'existence d'un chemin eulérien

fonction a_chemin_eulerien(G)

// retourne vrai ssi le graphe (simple) G admet un chemin eulérien
si G n'est pas connexe
  retourner faux
nb_impair $\leftarrow$ 0 // pour compter le nombre de sommets de degré impair
pour tout sommet s de G
  si cardinal(voisins(G,s)) est impair
    nb_impair $\leftarrow$ nb_impair + 1
retourner (nb_impair $\in \{0, 2\}$) // retourne le résultat du test

Implémentation Python :

```
def a_chemin_eulerien(G): # graphe -> booléen
    if not est_connexe(G):
        return False
    nb_impair = 0
    for s in sommets(G):
        if len(voisins(G,s))%2==1:
            nb_impair = nb_impair + 1
    return nb_impair in {0,2}
```

Parcours d'un graphe

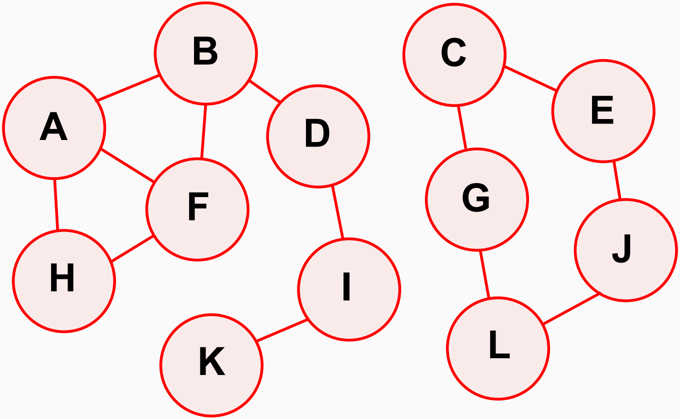
Beaucoup d'algorithmes évolués sur les graphes nécessitent d'effectuer un parcours du graphe.

Comme pour les arbres, essentiellement deux types de parcours :

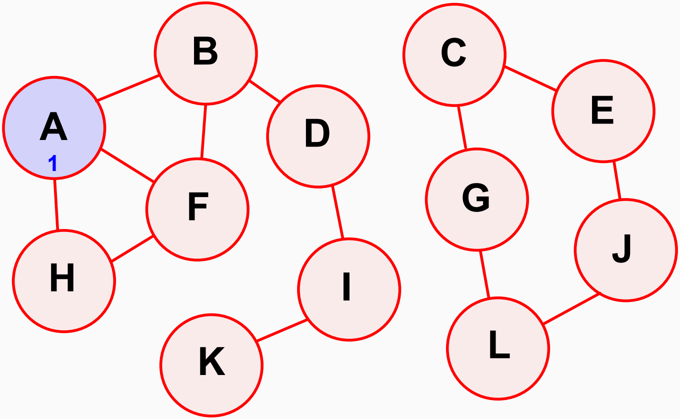
- le parcours en profondeur : on explore récursivement les voisins (non déjà visités) des voisins avant d'explorer les autres voisins
- le parcours en largeur : on explore tous les voisins (non déjà visités), puis on passe aux voisins des voisins

Dans les deux cas, il est nécessaire de **marquer** les sommets visités au fur et à mesure, par exemple en positionnant à Vrai une étiquette booléenne associée au sommet, ou bien en tenant à jour un ensemble de sommets marqués

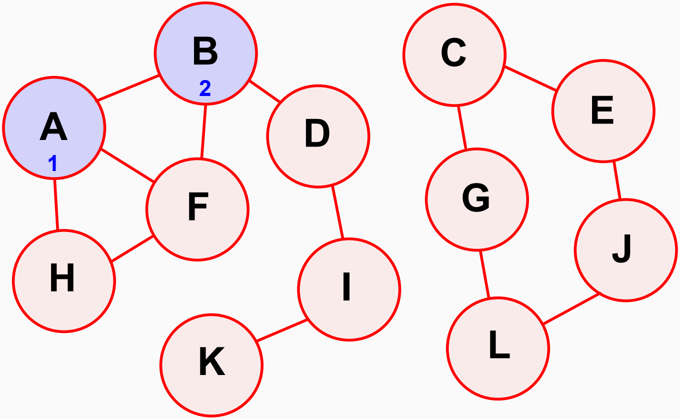
Parcours d’un graphe en profondeur



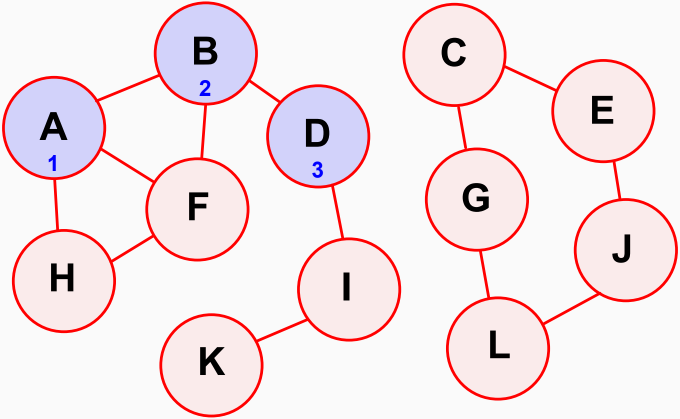
Parcours d’un graphe en profondeur

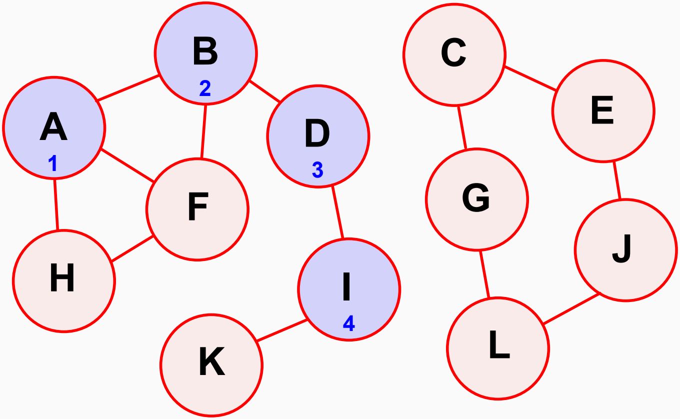


Parcours d'un graphe en profondeur

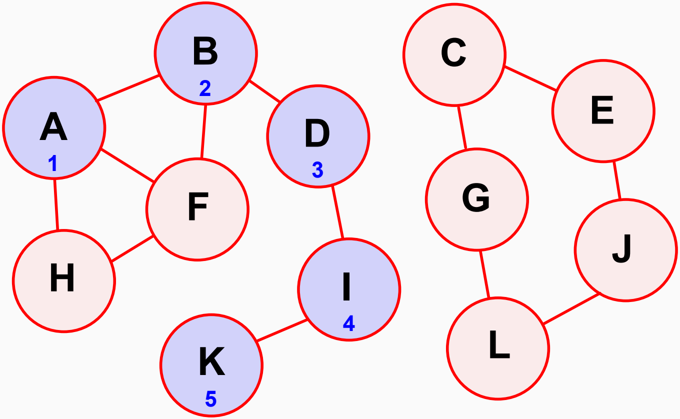


Parcours d'un graphe en profondeur

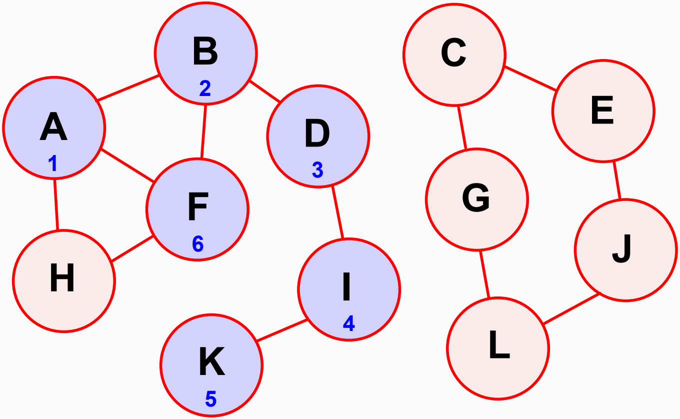



A
1
B
2
D
3
I
4
F
H
K
C
E
G
J
L

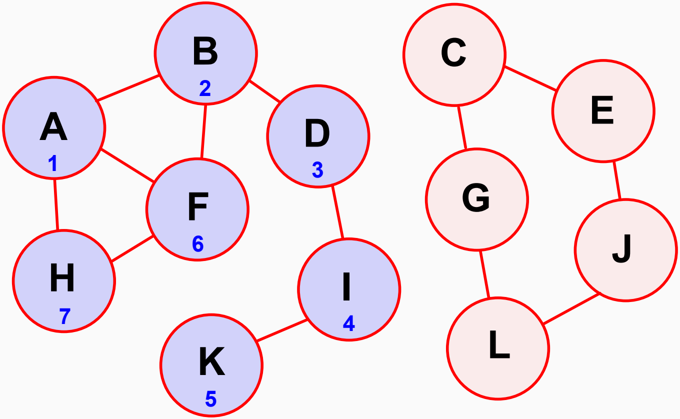
Parcours d'un graphe en profondeur



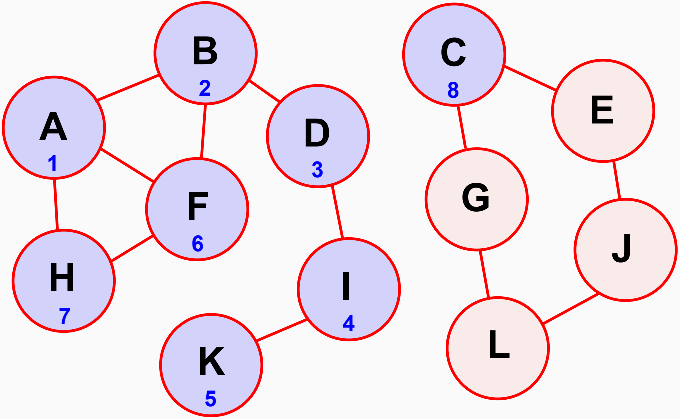
Parcours d'un graphe en profondeur



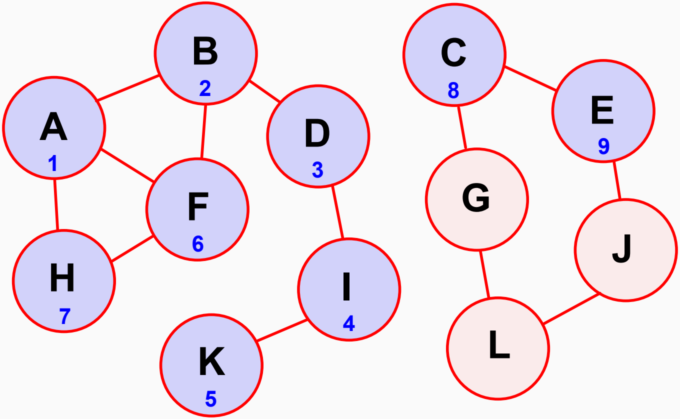
Parcours d'un graphe en profondeur



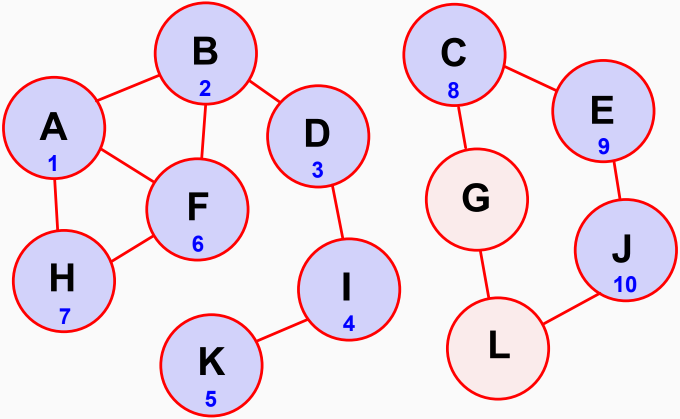
Parcours d’un graphe en profondeur



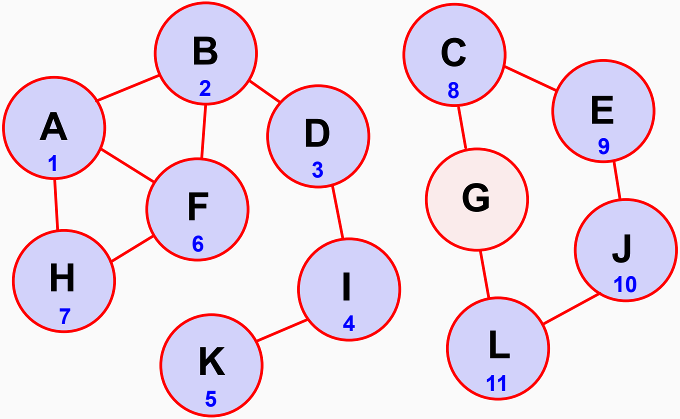
Parcours d'un graphe en profondeur



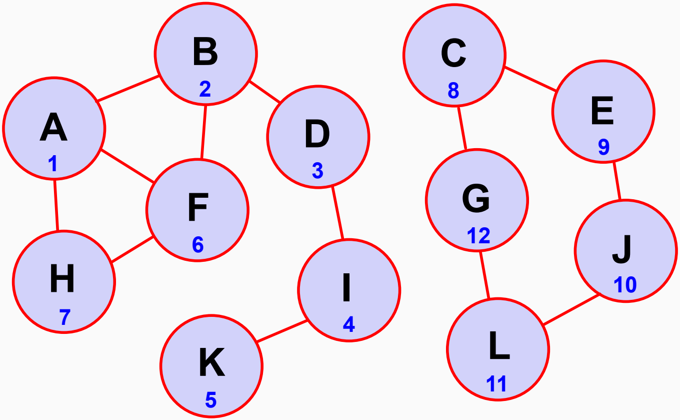
Parcours d'un graphe en profondeur



Parcours d'un graphe en profondeur



Parcours d'un graphe en profondeur



parcours en profondeur : A,B,D,I,K,F,H,C,E,J,L,G

il y a 2 composantes connexes

Parcours d'un graphe en profondeur

En partant du sommet s d'un graphe G et en explorant récursivement ses voisins, on visite la composante connexe de G contenant s (c'est-à-dire l'ensemble des sommets de G accessibles depuis s)

```
fonction explore(G,s)
    // explore la partie du graphe G accessible depuis le sommet s
    ... traiter s ... // dépend de l'algorithme à implémenter
    marquer s
    pout tout voisin t de s dans G
        si t n'est pas marqué
            explore(G,t)
```

Pour explorer complètement le graphe G, il faut appeler la fonction explore() pour toutes les composantes connexes de G

```
fonction parcours_profondeur(G)
    pour tout sommet s de G
        si s n'est pas marqué
            explore(G,s)
```

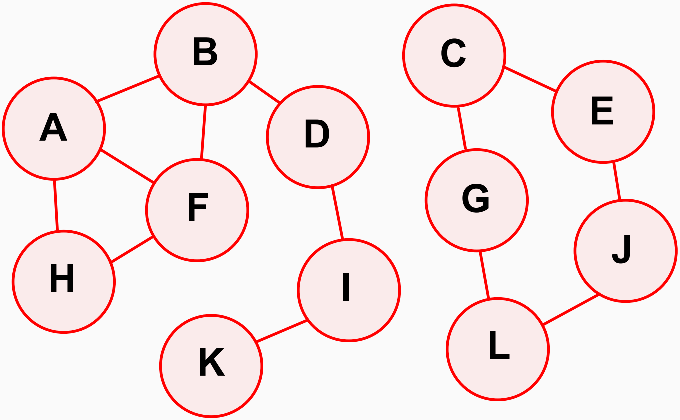
Application du parcours en profondeur : test de connexité

```
fonction explore(G,s,M)
  // explore la partie du graphe G accessible depuis le sommet s
  ajouter à l'ensemble M le sommet s // marque s
  pout tout voisin t de s dans G
    si t ∉ M // si t n'est pas marqué
      explore(G,t,M)

fonction nb_composantes_connexes(G)
  // retourne le nombre de composantes connexes du graphe G
  n ← 0 // nombre de composantes connexes
  M ← ∅ // ensemble des sommets marqués
  pour tout sommet s de G
    si s ∉ M // si s n'est pas marqué
      n ← n+1 // nouvelle composante connexe trouvée
      explore(G,s,M)// explore la composante connexe contenant s
  retourner n

fonction est_connexe(G)
  // retourne vrai si le graphe G est connexe, faux sinon
  retourner ( nb_composantes_connexes(G) = 1 )
```

Application du parcours en profondeur : test de connexité

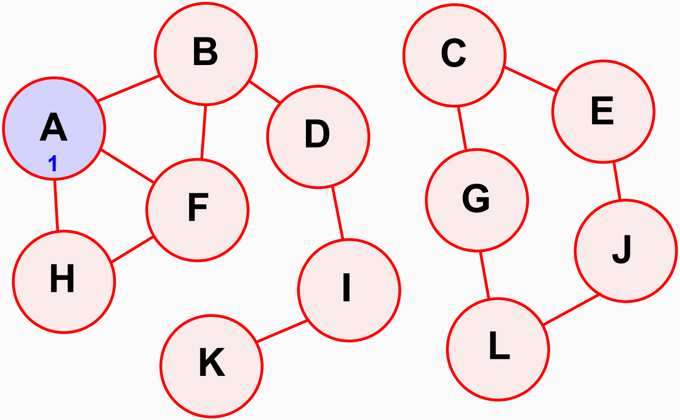
```
def explore(G,s,M):
    # graphe x sommet x ensemble de sommets (modifié) ->
    "parcours de la composante connexe de G contenant s"
    M.add(s) # on marque s
    for t in voisins(G,s):
        if t not in M: # si t n'est pas marqué
            explore(G,t,M)

def nb_composantes_connexes(G): # graphe -> entier
    "retourne le nombre de composantes connexes du graphe G"
    n = 0 # pour compter le nombre de composantes connexes
    M = set() # pour marquer les sommets
    for s in sommets(G):
        if s not in M:
            n = n + 1 # nouvelle composante connexe trouvée
            explore(G,s,M)
    return n

def est_connexe(G): # graphe -> booléen
    "retourne True si le graphe G est connexe, False sinon"
    return nb_composantes_connexes(G)==1
```

Parcours d'un graphe en largeur



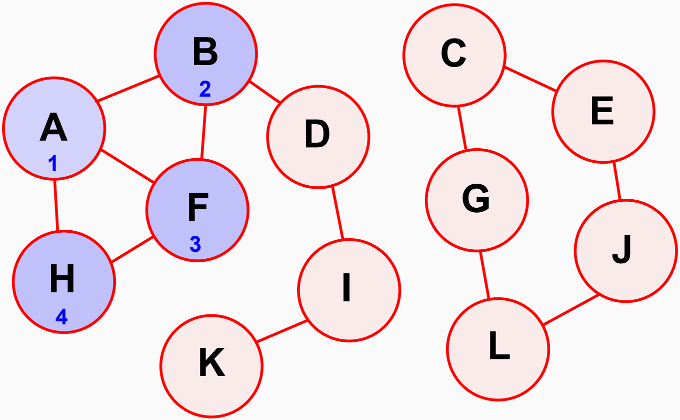
file vide $\longrightarrow$ $\boxed{\text{A}}$

Parcours d'un graphe en largeur



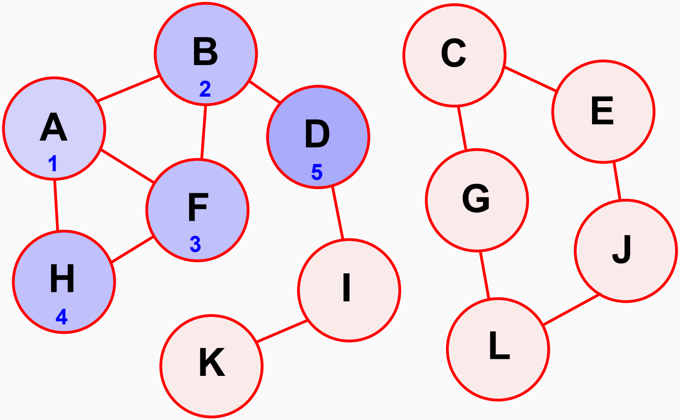
file : | A | ⟶ | H | F | B |

Parcours d'un graphe en largeur



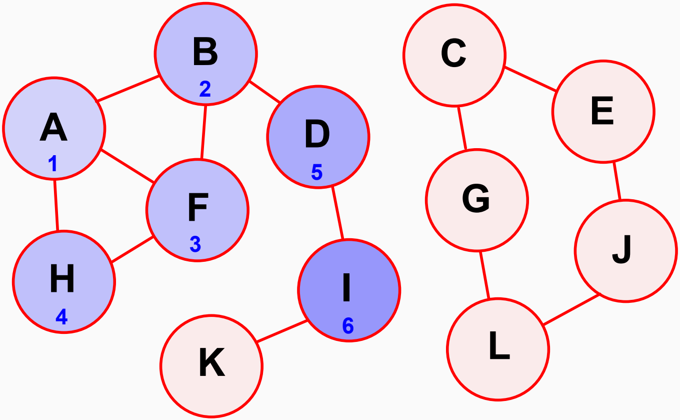
file : | H | F | B | $\longrightarrow$ | D |

Parcours d'un graphe en largeur



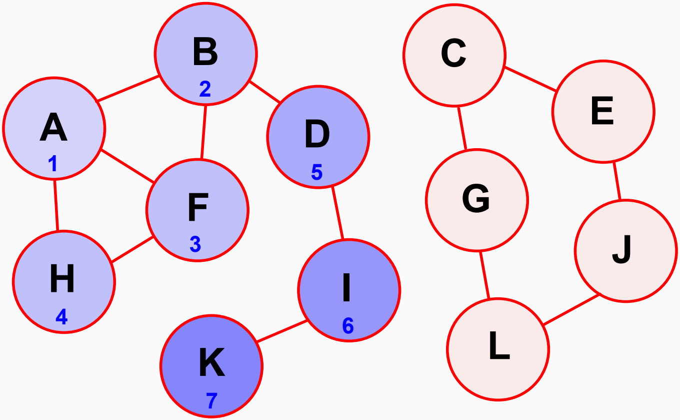
file : $\boxed{\text{D}} \longrightarrow \boxed{\text{I}}$

Parcours d'un graphe en largeur



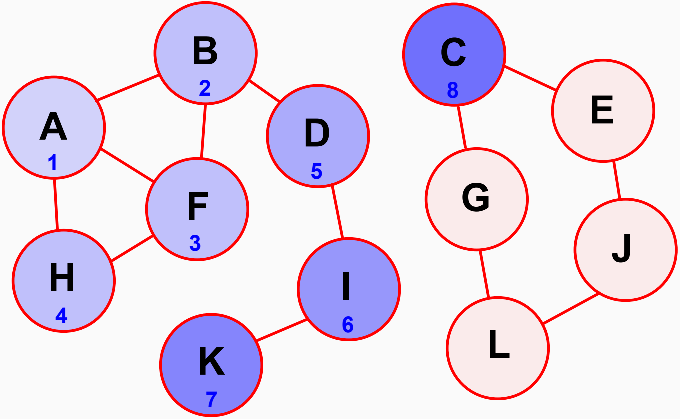
file : $\boxed{\text{I}} \longrightarrow \boxed{\text{K}}$

Parcours d'un graphe en largeur



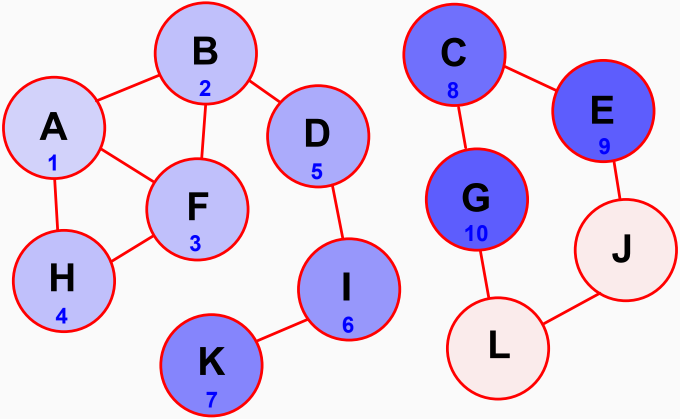
file : $\boxed{\text{K}} \longrightarrow \boxed{\text{C}}$

Parcours d'un graphe en largeur



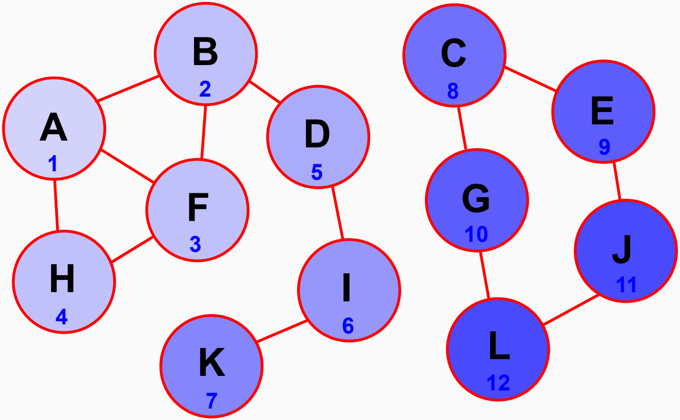
file : C ⟶ G E

Parcours d’un graphe en largeur



file : | G | E | $\longrightarrow$ | L | J |

Parcours d'un graphe en largeur



file : | L | J | $\longrightarrow$ vide

Parcours d'un graphe en largeur

Dans le **parcours d'un graphe en largeur**, on s'assure de traiter tous les voisins d'un sommet donné avant de considérer leurs propres voisins.

Pour cela, on ajoute au fur et à mesure les nouveaux voisins rencontrés dans une file (FIFO)

```
fonction parcours_largeur(G)// parcours du graphe G en largeur
  F ← file_vide()
  pour tout sommet s de G
    si s n'est pas marqué
      ajouter_élément(F,s)
      // explore la partie de G accessible depuis les sommets présents dans F
      tant que la file F n'est pas vide
        s ← extraire_élément(F)
        ... traiter s ... // dépend de l'algorithme à implémenter
        marquer s
        pout tout voisin t de s dans G // ajout dans F des voisins de s
          si t n'est pas marqué
            ajouter_élément(F,t)
```

Parcours d'un graphe en largeur : application

On appelle **distance entre deux sommets x et y** d'un graphe la longueur du plus petit chemin qui relie x à y (si x et y ne sont pas reliés, la distance est infinie).

Le parcours en largeur du graphe permet de calculer la distance de x à y, en parcourant successivement les sommets à distance 1, puis 2, etc. de x jusqu'à trouver y.

Remarque : Dans le cas d'un graphe pondéré, on définit la poids d'un chemin comme la somme des poids de ses arêtes, et la distance pondérée entre deux sommets comme le poids minimal d'un chemin qui les relie. Cette distance pondérée peut être calculée avec l'algorithme de Dijkstra (hors programme).

Calcul de la distance entre deux sommets d'un graphe

```
fonction distance(G,x,y)
    // retourne la distance de x à y dans G
    F ← file_vide()
    ajouter_élément(F,x)
    d ← 0
    tant que la file F n'est pas vide
        C ← file_vide() // couche suivante
        tant que F n'est pas vide
            s ← extraire_element(F)
            si s=y
                retourner d // on a atteint y → on retourne d
            si s n'est pas marqué
                marquer s
                pour tout voisin t de s dans G
                    si t n'est pas marqué
                        ajouter_élément(C,t)
        d ← d+1 // on incrémente la distance
        F ← C // on passe à la couche suivante
    retourner +∞ // pas de chemin de x à y
```

Calcul de la distance entre deux sommets d'un graphe

```
def distance(G,x,y):
    "retourne la distance de x à y, None si elle est infinie"
    F = file_vide()
    ajouter_element(F,x)
    M = set() # ensemble des sommets marqués (= visités)
    d = 0
    while not est_vide(F):
        C = file_vide()
        while not est_vide(F): # on explore F
            s = extraire_element(F)
            if s==y:
                return d # on a atteint y -> on retourne d
            if s not in M:
                M.add(s) # on marque s
                for t in voisins(G,s):
                    if t not in M:
                        ajouter_element(C,t)
        d = d+1
        F = C
    return None # pas de chemin de x à y -> distance infinie
```

Parcours d'un arbre en largeur

Pour les **arbres**, le parcours en largeur s'effectue de manière similaire, mais la fonction est plus simple, car :

- il est inutile de marquer les nœuds
- il n'y a qu'une composante connexe à visiter (issue de la racine)

```
fonction parcours_largeur(A)
    // parcours de l'arbre A en largeur
    F ← file_vide()
    ajouter_élément(F,racine(A))
    tant que la file F n'est pas vide
        N ← extraire_élément(F)
        ... traiter N ... // dépend de l'algorithme à implémenter
        pout tout enfant E de N // ajout dans la file F des enfants de N
            ajouter_élément(F,E)
```