	Python
test d'égalité	`==`
différent	`!=`
inférieur ou égal $\leq$	`<=`
division euclidienne	`//`
modulo	`%`
et	`and`
ou	`or`
négation	`not`

- On peut utiliser `if a % b == 0` pour savoir si `b` divise `a` (exemple : `if n % 2 == 0` pour savoir si `n` est pair).
- Ne pas confondre == (comparer deux valeurs, dans un `if` ou `while`) et = (modifier la valeur d'une variable).
- La variable modifiée est toujours à gauche du = : `a = b` modifie `a`.
- `L[i]` donne une erreur si `L[i]` n'existe pas :

```
L = []
L[0] = 0 # ERREUR !!!
L = [0] # faire ceci à la place
L = []
L.append(0) # ou ceci
```

- Ne pas écrire `if x == True` ou `if x == False` mais `if x` ou `if not x` (plus idiomatique).
- Si `L1` et `L2` sont des listes de tailles n_1 et n_2, `L1 + L2` donne en complexité $O(n_1 + n_2)$ une nouvelle liste contenant les éléments de `L1` suivis des éléments de `L2`.
- `n*L` duplique la liste `L` `n` fois (même chose que `L + L + ... + L`). Exemple : `[0]*4` donne `[0, 0, 0, 0]`.
- `[... for i in ...]` est une création de liste par compréhension. Par exemple, `[i**2 for i in range(5)]` donne `[0, 1, 4, 9, 16]` et est équivalent à :

```
L = []
for i in range(5):
    L.append(i**2)
```

- On peut aussi ajouter une condition dans une liste par compréhension : `[i**2 for i in range(5) if i % 2 == 0]` donne `[0, 4, 16]`.
- Opérations sur une matrice `M` (comme liste de listes) :

Python	Signification
`M[i][j]`	$m_{i,j}$ (élément ligne i, colonne j)
`M[i]`	$i$ème ligne
`len(M)`	nombre de lignes
`len(M[0])`	nombre de colonnes

- Créer une matrice $n \times p$ remplie de 0 :

```
M = [[0]*p for _ in range(n)]
```

Ou utiliser des boucles `for` et `append` :

```
M = []
for i in range(n):
    L = []
    for j in range(p):
        L.append(0)
    M.append(L)
```

Attention : le code ci dessous ne marche pas, car `M` a n fois la même ligne (modifier l'une modifie les autres).

```
M = []
L = []
for j in range(p):
    L.append(0)
for i in range(n):
    M.append(L) # M contient n fois la même liste L !!!
```

- Les types de base (`int`, `float`, `bool`...) sont copiés par défaut, contrairement aux `list` :

```
a = 3
b = a
b = 2 # ne modifie pas a
```

```
L1 = [3]
L2 = L1
L2[0] = 4 # modifie L1
```

De même lors du passage en argument d'une fonction :

```
def f(L):
    L[0] = 3
L1 = [2]
f(L1) # L1 est modifié
```

- Les indices commencent à partir de 0 : le premier élément est `L[0]`, le dernier `L[len(L) - 1]` (qui est obtenu aussi avec `L[-1]`).
- `L[i:j]` extrait de `L` une sous-liste des indices `i` à `j - 1`. On peut copier une liste avec `L[:]` ou `L.copy()`.
- Si `x` est une liste, un n-uplet, une chaîne de caractères ou un tableau numpy :
  - `x[i]` est le îème élément de `x`
  - `x[-i]` est le $i$ème élément en partant de la fin
  - `x[i:j]` extrait les éléments de `x` du $i$ème au $j$ème exclu (exemple : si)
  - `len(x)` est la taille de `x` En revanche, on ne peut pas modifier un n-uplet ou une chaîne de caractères (pas de `x[i] = ...` ou de `x.append(...)` dans ce cas).
- Éviter de faire plusieurs fois le même appel de fonction : stocker le résultat dans une variable à la place.
- Ne pas confondre indice et élément d'une liste : `for i in range(len(L))` parcourt les indices de `L` (`i` vaut 0, 1, 2, ..., `len(L) - 1`), alors que `for x in L` parcourt les éléments de `L` (`x` vaut `L[0]`, `L[1]`, `L[2]`, ..., `L[len(L) - 1]`).
- `for i in range(a, b, p)` parcourt les entiers de `a` à `b - 1` en allant de `p` en `p` (par défaut $a = 0$ et $p = 1$).
- Pour écrire une fonction récursive, il faut toujours un cas de base (qui ne fait pas appel à la fonction elle-même) et un cas récursif (qui fait appel à la fonction elle-même).

Pour les exemples, on considère une base de données avec 3 tables dont les schémas relationnels sont :

- film (id, titre, annee, directeur, budget, recette)
- acteur (id, nom)
- casting (id_film, id_acteur)

Une **clé** d'une table est un ensemble minimal d'attributs permettant d'identifier de façon unique chaque enregistrement.
La **clé primaire** d'une table est une clé dont on garantit l'unicité même après ajout dans la table.
Une **clé étrangère** est un attribut (ou ensemble d'attributs) faisant référence à une clé primaire d'une autre table.

Syntaxe générale de `SELECT`, dans cet ordre (`[...]` indiquant une commande optionnelle) :

```
SELECT [DISTINCT] expr1 [AS alias1], expr2, ...
FROM table1 [AS alias1], table2, ...
[WHERE condition]
[GROUP BY expr
[HAVING condition]]
[ORDER BY expr [DESC]]
[LIMIT entier
[OFFSET entier]]
```

- `SELECT [DISTINCT] expr1 [AS alias1], expr2, ...`
  Renvoie une table dont les colonnes correspondent à `expr1`, `expr2`...
  `expr1`, `expr2`... sont des expressions, pouvant contenir des attributs, calculs et fonctions. Si un attribut `attr` est ambigu (car il est le même dans 2 tables `t1` et `t2`), il faut le préfixer par son nom de table, par ex. `t1.attr`.
  $*$ est un raccourci pour selectionner toutes les colonnes.
  `AS` renomme une colonne pour, par exemple, y faire référence ensuite.
  `DISTINCT` supprime les doublons.
  Obtenir tous les acteurs (sans doublon) : `SELECT DISTINCT nom FROM acteur;`
  Films avec leur profit : `SELECT titre, recette - budget FROM film;`

- `FROM table1 [AS alias1], table2, ...`
  Tables d'où les valeurs sont sélectionnées.
  `table1, table2` est la table correspondant au produit cartésien de `table1` et `table2`.
  `table1 JOIN table2 ON colonne1 = colonne2` réalise la jointure de `table1` et `table2`, où la `colonne1` de `table1` est identifiée avec `colonne2` de `table2`. On peut mettre plusieurs `JOIN` à la suite.
  Tous les directeurs et acteurs ayant travaillé ensemble :

```
SELECT directeur, nom FROM film
JOIN casting ON film.id = id_film
JOIN acteur ON id_acteur = acteur.id
```

- `[WHERE condition]`
  Ne considère que les enregistrements vérifiant `condition`.
  `condition` peut contenir des attributs, calculs, `AND`, `OR`, `<`, `<=`, `!=`, `LIKE`, `IN`...
  Tous les directeurs qui sont aussi acteurs :

```
SELECT DISTINCT directeur FROM film, acteur
WHERE directeur = nom
```

- `[GROUP BY expr`
  `[HAVING condition]]`
  Regroupe tous les enregistrements ayant la même valeur `expr` en un seul enregistrement. Seuls les groupes vérifiant `condition` sont renvoyés.
  Les fonctions d'agrégations (dans un `SELECT` ou `HAVING`) s'appliquent alors pour chaque groupe : `COUNT(attribut)` (nombre d'enregistrements non `NULL`), `COUNT(*)` (nombre d'enregistrements), `SUM(attribut)`, `MAX(attribut)`, `AVG(attribut)` (moyenne), ...
  Nombre de films réalisés chaque année depuis 2000 :

```
SELECT annee, COUNT(*)
FROM film
WHERE annee >= 2000
GROUP BY annee;
```

  Directeurs ayant rapporté au moins 1 milliard :

```
SELECT directeur FROM film
GROUP BY directeur
HAVING SUM(recette) >= 1000000000
```

- `[ORDER BY expr [DESC]]`
  Trie les enregistrements selon `expr`, croissant par défaut (décroissant si `DESC` est utilisé).
  Acteurs triés par le nombre de films joués :

```
SELECT nom, COUNT(*) AS nb_films
FROM acteur JOIN casting ON acteur.id = id_acteur
            JOIN film ON film.id = id_film
GROUP BY nom
ORDER BY nb_films DESC
```

- `[LIMIT n`
  `[OFFSET p]]`
  Affiche seulement les n premiers enregistrements (en commençant à partir du $(p+1)$ème). Souvent utilisé après un `ORDER BY`.
  Deuxième film à plus gros budget :

```
SELECT titre FROM film
ORDER BY budget DESC
LIMIT 1 OFFSET 2;
```

- **Sous-requêtes** : il est possible d'utiliser un `SELECT` renvoyant une seule valeur à l'intérieur d'un autre `SELECT`, dans une condition ou un calcul. *Tous les acteurs du film à plus gros budget* :

```
SELECT nom FROM acteur
JOIN casting ON id_acteur = acteur.id
JOIN film ON id_film = film.id
WHERE titre = (SELECT titre FROM film
    ORDER BY budget DESC LIMIT 1)
```

- **Opérateurs ensemblistes** :
  Étant donné deux requêtes de la forme `SELECT ...`

renvoyant deux relations `table1` et `table2` de **même schéma relationnel**, il est possible d'obtenir leur union, intersection et différence avec `UNION`, `INTERSECT`, `MINUS`.
Exemple :

table1		
attr1	attr2	attr3
a1	a2	a3
b1	b2	b3

table2		
attr1	attr2	attr3
a1	a2	a3
c1	c2	c3

attr1	attr2	attr3
a1	a2	a3
b1	b2	b3
c1	c2	c3

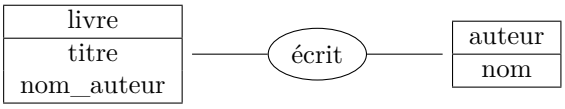
Résultat de
`SELECT * FROM table1 UNION SELECT * FROM table2;`

attr1	attr2	attr3
b1	b2	b3

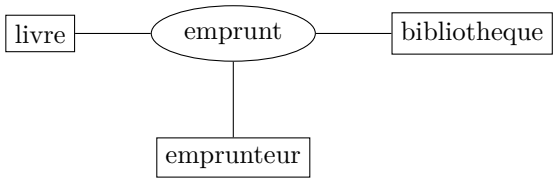
Résultat de
`SELECT * FROM table1 MINUS SELECT * FROM table2`

Modèle entité-association

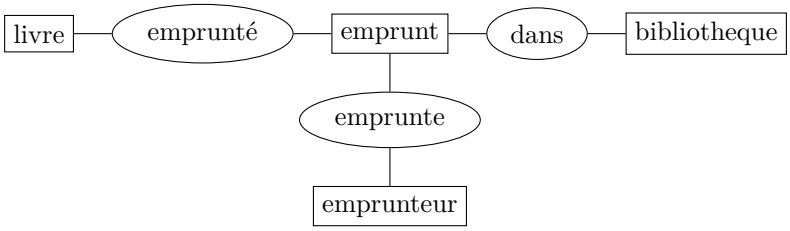
- Une **entité** est un ensembles d'objets similaires que l'on souhaite stocker.
  Exemple : Livre, auteur...
- Une **association** (ou : **relation**) est une relation entre plusieurs entités.
  Une association est binaire si elle met en relation deux entités.
  Exemple : Un auteur *écrit* un livre.
- Représentation sous forme de diagramme :



- Une relation n-aire peut être transformée en relation binaire en introduisant une nouvelle entité pour la relation.
  Exemple :

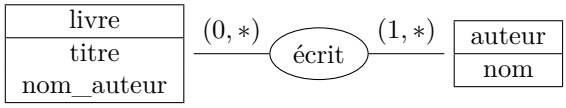


Une relation 3-aire



3 relations binaires

- On peut spécifier le lien entre une entité et une association avec un couple (n, p) indiquant le nombre minimum et maximum de fois que l'entité peut apparaître dans l'association ($p = *$ s'il n'y a pas de maximum).
  Exemples :
  - Un livre a été écrit par au moins une personne, sans limite supérieure. D'où la cardinalité $(1, *)$ pour le lien entre l'entité livre et l'association « écrit ».
  - Une personne peut avoir écrit un nombre quelconque de livre. D'où la cardinalité $(0, *)$.
  - Si on suppose qu'une personne peut emprunter au plus 5 livres, alors le lien entre l'entité personne et l'association « emprunt » est de cardinalité $(0, 5)$.



- Types possibles d'association entre deux entités :
  - $1-1$ (*one-to-one*) : La borne supérieure vaut 1 pour les 2 entités.
    Exemple : L'association « dirige » est de type $1-1$ pour directeur_bibliotheque et bibliotheque.
  - $1-*$ (*one-to-many*) : La borne supérieure vaut 1 pour une entité et $*$ pour l'autre.
    Exemple : Chaque livre est écrit par un unique auteur, mais chaque auteur a pu écrire plusieurs livres.
  - $*-*$ (*many-to-many*) : La borne supérieure vaut $*$ des deux côtés.
    Exemple : L'association « est de type » entre la table des pokémons et des types est de type $*-*$ (à chaque pokémon peut correspondre plusieurs types et plusieurs pokémons peuvent avoir le même type).
- Pour concevoir une base de donnée :
  - Utiliser une table par entité.
  - Pour chaque association entre a et b :
    - Si association $1-1$: Fusionner les tables a et b.
    - Si association $1-*$: Ajouter un attribut (clé étrangère) à b faisant référence à un a.
    - Si association $*-*$: Ajouter une table ayant 2 clé étrangère pour faire référence à a et b.

- Une fonction f est **récursive** si elle s'appelle elle-même. Elle est composée de :
  - Un (ou plusieurs) **cas de base** où f renvoie directement une valeur, sans appel récursif.
  - Un (ou plusieurs) **appel récursif** à f avec des arguments plus petits que ceux de l'appel initial, qui garantit de se ramener au cas de base.

  Exemple : Calcul de $n! = n \times (n-1)!$.

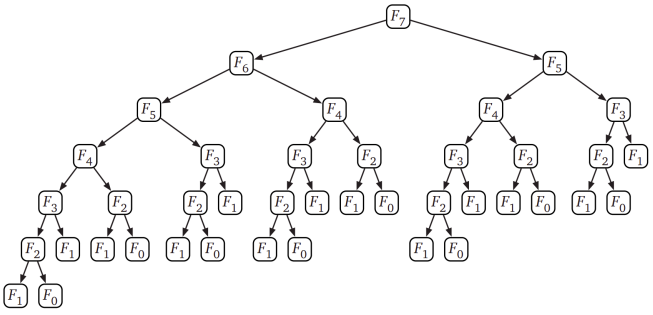
```python
def fact(n):
    if n == 0: # cas de base
        return 1
    return n*fact(n - 1) # appel récursif
```

  Attention : une fonction récursive peut ne pas terminer (appels récursifs infinis), de même qu'un `while` peut faire boucle infinie.

- La fonction suivante calcule les termes de la suite de Fibonacci ($u_0 = u_1 = 1, u_n = u_{n-1} + u_{n-2}$) :

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

  Cependant, la complexité est très mauvaise (exponentielle)... En effet, si $C(n)$ = complexité de `fibo(n)`, alors $C(n) = \underbrace{C(n-1)}_{fibo(n-1)} + \underbrace{C(n-2)}_{fibo(n-2)} + K$ ce qui est une équation récurrente linéaire d'ordre 2 (du même type que celle vérifiée par la suite de Fibonacci) que l'on peut résoudre pour trouver $C(n) \sim Ar^n$.
  Le soucis vient du fait que le même sous-problème est résolu plusieurs fois, ce qui est inutile et inefficace.



Appels récursifs de la version naïve de `fibo(7)`

- La **programmation dynamique** stocke les résultats des sous-problèmes pour éviter de les calculer plusieurs fois.

```python
def fibo(n):
    L = [1, 1] # L[i] = ième terme de Fibonacci
    for i in range(n - 1):
        L.append(L[i] + L[i + 1]) # récurrence
    return L[n]
```

- Pour résoudre un problème de programmation dynamique :
  1. Chercher une équation de récurrence. Souvent, cela demande d'introduire un paramètre.
  2. Déterminer le(s) cas de base(s). Il faut qu'appliquer plusieurs fois l'équation de récurrence ramène à un cas de base.
  3. Stocker en mémoire (dans une liste, matrice ou dictionnaire) les résultats des sous-problèmes pour éviter de les calculer plusieurs fois.

- Exemple : Calcul de $\binom{n}{k}$ en utilisant la formule de Pascal $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ et les cas de base $\binom{n}{0} = 1$ et $\binom{n}{0} = 0$ si $n \neq 0$.
  On utilise une matrice `M` telle que `M[i][j]` contienne $\binom{i}{j}$.

```python
def binom(n, k):
    M = [[0]*(k + 1) for _ in range(n + 1)]
    for i in range(0, n + 1):
        M[i][0] = 1
    for i in range(1, n + 1):
        for j in range(1, k + 1):
            M[i][j] = M[i - 1][j - 1] + M[i - 1][j]
    return M[n][k]
```

- La **mémoïsation** est similaire à la programmation dynamique mais en utilisant une fonction récursive plutôt que des boucles.
  Pour éviter de résoudre plusieurs fois le même problème (comme pour Fibonacci), on mémorise (dans un tableau ou un dictionnaire) les arguments pour lesquelles la fonction récursive a déjà été calculée.

- Version mémoïsée du calcul de la suite de Fibonacci :

```python
def fibo(n):
    d = {} # d[k] contiendra le kème terme de la suite
    def aux(k):
        if k == 0 or k == 1:
            return 1
        if k not in d:
            d[k] = aux(k - 1) + aux(k - 2)
        return d[k]
    return aux(n)
```

- Mémoïsation du calcul de coefficient binomial :

```python
def binom(n, k):
    d = {}
    def aux(i, j):
        if j == 0: return 1
        if i == 0: return 0
        if (i, j) not in d:
            d[(i, j)] = aux(i - 1, j - 1) + aux(i - 1, j)
        return d[(i, j)]
    return aux(n, k)
```

- Problème du sac à dos.
  Entrée : un sac à dos de capacité c, des objets $o_1, ..., o_n$ de poids $w_1, ..., w_n$ et valeurs v_1, ..., v_n. On suppose que les poids sont strictement positifs.
  Sortie : la valeur maximum que l'on peut mettre dans le sac.
  Soit $dp[i][j]$ la valeur maximum que l'on peut mettre dans un sac de capacité i, en ne considérant que les objets o_1, ..., o_j.

$$dp[i][0] = 0$$

$$dp[i][j] = \max(\underbrace{dp[i][j-1]}_{\text{sans prendre } o_j} , \underbrace{dp[i-w_j][j-1] + v_j}_{\text{en prenant } o_j, \text{si } i-w_j \geq 0})$$

Résolution par programmation dynamique :

```python
def knapsack(c, w, v):
    """
    Renvoie la valeur maximum que l'on peut mettre
    dans un sac à dos de capacité c.
    Le ième objet a pour poids w[i] et valeur v[i].
    """
    n = len(w) # nombre d'objets
    dp = [[0]*(n + 1) for i in range(c + 1)]
    # dp[i][j] = valeur max dans un sac de capacité i
    # où j est le nombre d'objets autorisés
    for i in range(1, c + 1):
        for j in range(1, n + 1):
            if w[j - 1] <= i:
                x = v[j - 1] + dp[i - w[j - 1]][j - 1]
                dp[i][j] = max(dp[i][j - 1], x)
            else:
                dp[i][j] = dp[i][j - 1]
    return dp[c][n]
```

Résolution par mémoïsation :

```python
def knapsack_memo(c, w, v):
    dp = {}
    def aux(i, j):
        if i == 0 or j == 0:
            return 0
        if (i, j) not in dp:
            dp[(i, j)] = aux(i, j - 1)
            if w[j - 1] <= i:
                x = v[j - 1] + aux(i - w[j - 1], j - 1)
                dp[(i, j)] = max(dp[(i, j)], x)
        return dp[(i, j)]
    return aux(c, len(w))
```

Algorithmes de classification

- Un algorithme de **classification** consiste à associer à chaque **donnée** une **classe** (ou : étiquette).
  Exemple : à chaque image, on veut associer l'objet représenté par l'image.

- On distingue deux types d'algorithmes de classification :
  - **Supervisé** : on possède des **données d'entraînement** pour lesquelles on connaît les classes. On veut ensuite prédire les classes de nouvelles données (**données de test**).
    Exemple : algorithme des plus proches voisins.
  - **Non supervisé** : pas de données d'entraînement.
    Exemple : algorithme des k-moyennes.

- Les algorithmes d'apprentissage ont besoin d'une notion de distance entre les données. Pour cela, on se ramène à $\mathbb{R}^k$. Ainsi, si une donnée de voiture possède une vitesse maximum de 200 km/h, consomme 10 litres au 100 km et pèse 1500 kg, on peut la représenter par le vecteur $\begin{pmatrix} 200 \\ 10 \\ 1500 \end{pmatrix}$.
  Pour une image : on passe d'une matrice de pixels avec n lignes, p colonnes à un vecteur de taille np.

- On représente classiquement l'ensemble des données (donc de vecteurs de $\mathbb{R}^p$) par une matrice `X` dont les lignes sont les données et les colonnes sont les attributs (coordonnées des vecteurs).

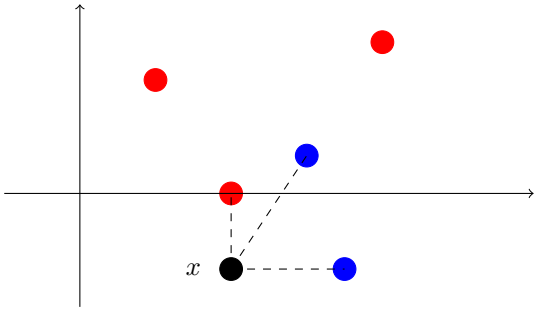
- On peut utiliser la distance euclidienne entre deux données : $d(x,y) = \sqrt{\sum_{i=1}^{p}(x_i - y_i)^2}$

```python
def d(x, y):
    s = 0
    for i in range(len(x)):
        s += (x[i] - y[i]) ** 2
    return s ** 0.5
```

- Pour éviter que les données ne soient trop influencées par les attributs qui ont des valeurs plus grandes, on peut standardiser (ou : normaliser) les données. On soustrait à chaque attribut sa moyenne et on divise par l'écart-type.

Algorithme des k plus proches voisins

- Soit $k \in \mathbb{N}$. L'**algorithme des k plus proches voisins** prédit la classe d'une nouvelle donnée x de la façon suivante:
  1. Calculer les distances de x à toutes les données d'entraînement.
  2. Trouver les k données d'entraînement les plus proches de x (en termes de distance).
  3. Trouver la classe majoritaire $c \in Y$ parmi ces k données les plus proches de x.
  4. Prédire que x est de classe c.



La classe de x est prédite comme étant bleue (ici, $k = 3$ et il y a deux classes : bleu et rouge)

- Exercice : Écrire une fonction `voisins(x, X, k)` renvoyant la liste des k plus proches voisins de `x` dans `X`.

```python
def voisins(x, X, k):
    I = [] # indices des k plus proches voisins dans X
    for i in range(k): # ajout du ième minimum
        jmin = 0
        for j in range(len(X)):
            if d(x, X[j]) < d(x, X[jmin]) and j not in I:
                jmin = j
        I.append(jmin)
    return I
```

- Exercice : Écrire une fonction `maj(L)` renvoyant l'élément le plus fréquent de la liste `L`, en complexité linéaire.

```python
def maj(L):
    C = {} # C[e] = nombre d'occurrences de e dans L
    for e in L:
        if e in C:
            C[e] += 1
        else:
            C[e] = 1
    kmax = L[0]
    for k in C:
        if C[k] > C[kmax]:
            kmax = k
    return kmax
```

Complexité : $O(n)$, où n est la taille de `L`.

- Exercice : Écrire une fonction `knn(x, X, Y, k)` renvoyant la classe prédite par l'algorithme des k plus proches voisins pour la donnée `x` et les données d'entraînement `X` et `Y`.

```python
def knn(x, X, Y, k):
    """ Prédit la classe de x avec l'algorithme KNN
    x : nouvelle donnée
    X : données d'entraînement
    Y : étiquettes des données d'entraînement
    k : nombre de voisins à considérer
    """
    V = voisins(x, X, k)
    return maj([Y[i] for i in V])
```

- Supposons que l'on possède des données X avec des éti-

quettes Y et qu'on veuille savoir si KNN est un bon classifieur pour ces données.

Pour cela, on partitionne les données en deux ensembles :

- Ensemble d'entraînement X_{train} (de classes Y_{train}) : données parmi lesquelles on va chercher les k plus proches voisins.
- Ensemble de test X_{test} (de classes Y_{test}) : données utilisées pour évaluer le modèle, en comparant les classes prédites par KNN avec les classes réelles.

- La **précision** d'un modèle est la proportion de données de test bien classées par rapport au nombre total de données.

```
def precision(k):
    n = len(X_test)
    p = 0
    for i in range(n):
        if predict(X_test[i], k) == Y_test[i]:
            p += 1
    return p/n
```

- La **matrice de confusion** est une matrice carrée dont les lignes et les colonnes sont les classes possibles. La case (i, j) contient le nombre de données de test de classe i qui ont été prédites comme appartenant à la classe j.
Exemple : Dans la matrice suivante, on voit que 21 données de classe 0 ont été prédites comme appartenant à la classe 0, 2 données de classe 1 ont été prédites comme appartenant à la classe 2...

```
array([[21,  0,  0],
       [ 0, 29,  1],
       [ 0,  2, 23]])
```

- Pour choisir la valeur de k dans l'algorithme des k plus proches voisins, on peut afficher la précision en fonction de k pour choisir la valeur de k qui donne la meilleure précision.

Algorithme des k moyennes

- Le **centre** (ou : **isobarycentre**) d'un ensemble de vecteurs $x_1, \dots, x_n$ est le vecteur $\overline{x} = \dfrac{1}{n}\sum_{i=1}^{n} x_i$.
Exercice : Écrire une fonction `centre(X)` renvoyant le centre de la liste de vecteurs `X`.

```
def centre(X):
    n, p = len(X), len(X[0])
    x = [0] * p
    for i in range(n):
        for j in range(p):
            x[j] += X[i][j]
    for j in range(p):
        x[j] /= n
    return x
```

- La variance (ou : moment d'inertie) $V(X)$ d'un ensemble de vecteur X est définie par

$$V(X) = \sum_{x \in X} d(x, \overline{X})^2$$

La variance mesure la variation par rapport à la moyenne : plus $V(X)$ est petit, plus les vecteurs de X sont proches du barycentre $\overline{X}$.
Objectif : trouver un partitionnement (*clustering*) de X en k sous-ensembles $X_1, \dots, X_k$ (classes ou *clusters*) minimisant l'inertie I : $I = \sum_{i=1}^{k} V(X_i)$
Dit autrement : on veut associer à chaque donnée x une classe k telle que l'inertie I soit minimum.
Plus l'inertie est petite, plus les données sont proches du centre de leur classe et plus le partitionnement est bon.

Algorithme des k-moyennes

Objectif : partitionnner X en classes X_1, ..., X_k.

1. Soit $c_1, ..., c_k$ des vecteurs choisis aléatoirement.
2. Associer chaque donnée x à la classe X_i telle que $d(x, c_i)$ soit minimale.
3. Recalculer les centres des classes $c_i = \overline{X_i}$.
4. Si les centres ont changé, revenir à l'étape 2.

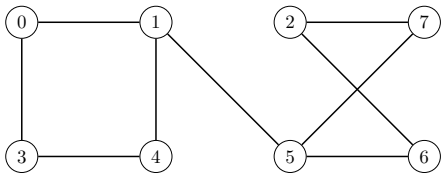
Attention : dans l'algorithme des k-moyennes, k est le nombre de classes alors que dans l'algorithme des plus proches voisins, k est le nombre de voisins.

```
def calculer_centres(classes):
    centres = []
    for i in range(len(classes)):
        centres.append(centre(classes[i]))
    return centres

def plus_proche(x, centres):
    imin = 0
    for i in range(len(centres)):
        if d(x, centres[i]) < d(x, centres[imin]):
            imin = i
    return imin

def calculer_classes(X, centres):
    classes = [[] for i in range(len(centres))]
    for x in X:
        classes[plus_proche(x, centres)].append(x)
    return classes

def kmoyennes(X, centres):
    centres2 = None
    while centres != centres2:
        centres2 = centres
        classes = calculer_classes(X, centres2)
        centres = calculer_centres(classes)
    return classes
```

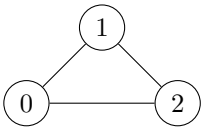
- Un **puit** t dans un graphe orienté est un sommet qui n'a pas de successeur (il n'existe pas d'arête allant vers t).

- Théorème : Tout graphe orienté acyclique G possède un puit.
  Preuve : Soit v un sommet de G. On fait partir un chemin depuis v en se déplaçant vers un successeur à chaque étape. Comme G est acyclique, ce chemin ne peut pas revenir deux fois au même sommet. Comme le nombre de sommets est fini, ce chemin doit forcément arriver à un puit.

- Un graphe $G = (V, E)$ est **biparti** s'il existe une partition $V = V_A \sqcup V_B$ telle que toute arête de E a une extrémité dans V_A et une extrémité dans V_B.



Graphe biparti, avec une partition $V_A = \{0, 4, 5, 2\}$ et $V_B = \{1, 3, 6, 7\}$

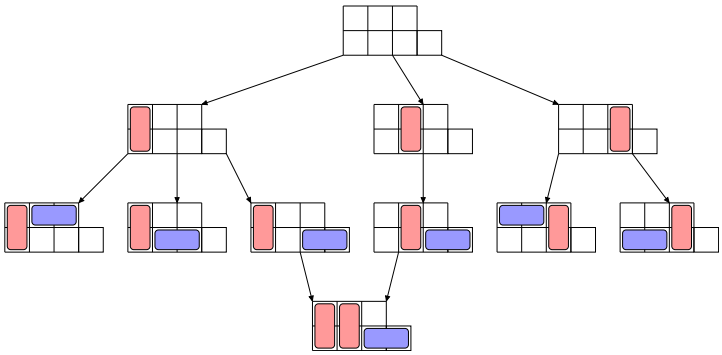


Graphe non biparti

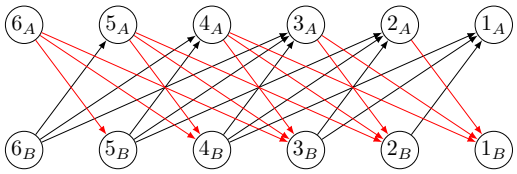
- On s'intéresse à un jeu à deux joueurs (Alice et Bob), qui se joue chacun son tour. On suppose qu'Alice commence. Un joueur a perdu lorsqu'il n'a plus de coup possible.
  Remarque : Les règles peuvent changer suivant le jeu auquel on joue.

- Exemples de jeux :
  - Le jeu de Nim : il y a n allumettes. Chaque joueur peut retirer 1, 2 ou 3 allumettes. Le joueur qui retire la dernière allumette a perdu.
  - Le jeu du domineering est un jeu de plateau où Alice place un domino vertical et Bob un domino horizontal. Un joueur qui ne peut plus jouer perd.

- Le **graphe des configurations** d'un jeu est un graphe orienté dont les sommets sont les configurations possibles du jeu et les arêtes sont les coups possibles.



Graphe des configurations d'un jeu de domineering



Graphe des configurations d'un jeu de Nim avec, initialement $n = 6$. Pour déterminer cmplètement une configuration, on indique le joueur dont c'est le tour.

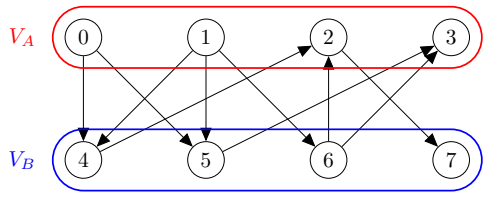
- Soit $G = (V, E)$ un graphe biparti acyclique, avec $V = V_A \sqcup V_B$.
  - Le jeu commence en un **sommet initial** $v \in A$.
  - Une **partie** est un chemin commençant en v dont les arêtes sont choisies alternativement par Alice et Bob.
  - L'ensemble P_A des puits de V_A sont les situations où Alice perd.
  - Une **stratégie** pour Alice est une fonction $f : V_A \setminus P_A \to V_B$ telle que $\forall v \in V_A$, $(v, f(v)) \in E$.
    Une stratégie consiste donc à choisir un coup à jouer pour chaque configuration possible.
  - Une **stratégie gagnante** pour Alice est une stratégie f qui permette à Alice de gagner, quel que soit la stratégie de Bob.
  - $v \in V$ est une **position gagnante** pour Alice si elle possède une stratégie gagnante pour une partie qui commence en v.
  - L'**attracteur** A d'Alice est l'ensemble des position gagnantes pour Alice.

  Définitions sont similaires pour Bob en échangeant A et B.

- On peut déterminer l'attracteur A d'Alice de proche en proche, en calculant l'ensemble A_k des positions gagnantes pour Alice en k coups :
  - $A_0 = P_B$ (gagnants pour Alice).
  - Si k est pair : $A_{k+1} = \{u \in V_A \mid \exists (u, v) \in E, v \in A_k\}$
  - Si k est impair : $A_{k+1} = \{u \in V_B \mid \forall (u, v) \in E, v \in A_k\}$

S'il y a n sommets, un chemin possède au plus $n-1$ arêtes d'où : $A = \bigcup_{k=0}^{n-1} A_k$.

Exemple :



$A_0 = \{7\}$, $A_1 = \{2\}$, $A_2 = \{4\}$, $A_3 = \{0, 1\}$.
Ainsi l'attracteur A d'Alice est $\{0, 1, 2, 4, 7\}$.
On peut aussi en déduire une stratégie gagnante qui consiste à jouer sur un attracteur si possible : $f(0) = 4$, $f(1) = 4$, $f(2) = 7$.

- (Formulation récursive équivalente à la précédente) Un sommet v est un attracteur dans l'un des trois cas suivants :
  - $v \in P_B$.
  - $v \in V_A$ et il existe un attracteur w tel que $(v, w) \in E$.
  - $v \in V_B$ et pour tout $(v, w) \in E$, w est un attracteur.

D'où la fonction récursive (par mémoïsation pour éviter des calculs inutiles), où `G` est représenté par dictionnaire d'adjacence et `fA` est une fonction indiquant si un sommet appartient à V_A :

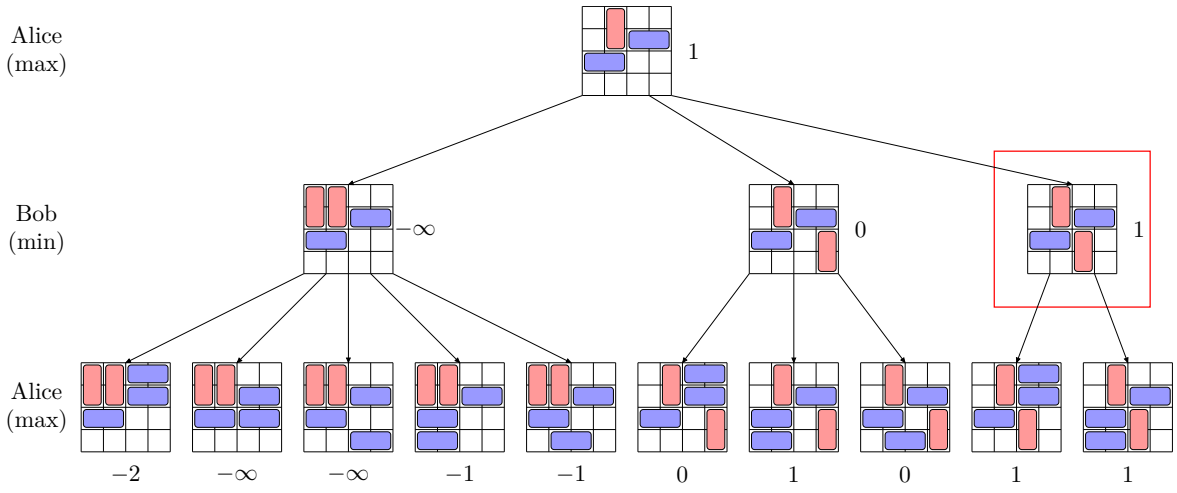
```
def attracteurs(G, fA):
    d = {} # d[v] = True si v est un attracteur
    def aux(v): # détermine si v est un attracteur
        if v not in d:
            succ = [aux(w) for w in G[v]]
            if len(G[v]) == 0:
                d[v] = not fA(v)
            elif fA(v):
                # test s'il existe (v, w) ∈ E avec w attracteur
                d[v] = False
                for w in G[v]:
                    if aux(w):
                        d[v] = True
            else:
                # test si pour tout (v, w) ∈ E, w est attracteur
                d[v] = True
                for w in G[v]:
                    if not aux(w):
                        d[v] = False
        return d[v]
    return [v for v in G if aux(v)]
```

- Une **heuristique** pour un jeu est une fonction qui à une configuration associe une valeur dans $\mathbb{R}$ et qui estime à quel point la configuration v est favorable à un joueur : plus $h(v)$ est grand, plus v est favorable à Alice et inversement.
  Exemple : dans le jeu du domineering, on peut utiliser $h(v)$ = nombres de possibilités pour Alice - nombres de possibilités pour Bob.
  Attention : Aussi bien dans A* que min-max, utiliser une heuristique permet d'accélérer la recherche mais le résultat n'est pas forcément optimal.

- L'algorithme de calcul des attracteurs est trop lent si le graphe des configurations est trop grand (échec, go...).
  L'algorithme **min-max** permet d'accélérer la recherche avec une heuristique et en ne parcourant que les configurations atteignable en au plus p coups. Il donne une valeur à chaque sommet de l'arbre de proche en proche :
  - La valeur des sommets à profondeur p et ceux sans successeurs.
  - La valeur des sommets à profondeur $p-1$ est le maximum (si Alice soit jouer) ou le minimum (si Bob doit jouer) des valeurs des successeurs.
  - Ainsi de suite, jusqu'à calculer la valeur de la racine.

```
def minmax(s, h, v, p, j):
    succ = [minmax(s, h, w, p - 1, 1 - j) for w in s(v, j)]
    if succ == [] or p == 0:
        return h(v)
    if j == 0:
        return max(succ)
    else:
        return min(succ)
```

Une fois la valeur de chaque configuration calculée, Alice choisit le coup dont la valeur est la plus élevée (le plus favorable).



Arbre min-max rempli de bas en haut, avec le coup optimal pour Alice entouré.

2 Programme du second semestre

2.1 Méthodes de programmation et analyse des algorithmes

On formalise par des leçons et travaux pratiques le travail entrepris au premier semestre concernant la discipline et les méthodes de programmation.
Même si on ne prouve pas systématiquement tous les algorithmes, on dégage l'idée qu'un algorithme doit se prouver et que sa programmation doit se tester.

Notions	**Commentaires**
Instruction et expression. Effet de bord.	On peut signaler par exemple que le fait que l'affectation soit une instruction est un choix des concepteurs du langage Python et en expliquer les conséquences.
Spécification des données attendues en entrée, et fournies en sortie/retour.	On entraîne les étudiants à accompagner leurs programmes et leurs fonctions d'une spécification. Les signatures des fonctions sont toujours précisées.
Annotation d'un bloc d'instructions par une précondition, une postcondition, une propriété invariante.	Ces annotations se font à l'aide de commentaires.
Assertion.	L'utilisation d'assertions est encouragée par exemple pour valider des entrées. La levée d'une assertion entraîne l'arrêt du programme. Ni la définition ni le rattrapage des exceptions ne sont au programme.
Explicitation et justification des choix de conception ou programmation.	Les parties complexes de codes ou d'algorithmes font l'objet de commentaires qui l'éclairent en évitant la paraphrase. Le choix des collections employées (par exemple, liste ou dictionnaire) est un choix éclairé.
Terminaison. Correction partielle. Correction totale. Variant. Invariant.	La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête, la correction est totale si elle est partielle et si l'algorithme termine. On montre sur plusieurs exemples que la terminaison peut se démontrer à l'aide d'un variant de boucle. Sur plusieurs exemples, on explicite, sans insister sur aucun formalisme, des invariants de boucles en vue de montrer la correction des algorithmes.
Jeu de tests associé à un programme.	Il n'est pas attendu de connaissances sur la génération automatique de jeux de tests ; un étudiant doit savoir écrire un jeu de tests à la main, donnant à la fois des entrées et les sorties correspondantes attendues. On sensibilise, par des exemples, à la notion de partitionnement des domaines d'entrée et au test des limites.
Complexité.	On aborde la notion de complexité temporelle dans le pire cas en ordre de grandeur. On peut, sur des exemples, aborder la notion de complexité en espace.

2.2 Représentation des nombres

On présente sans formalisation théorique les enjeux de la représentation en mémoire des nombres. Ces notions permettent d'expliquer certaines difficultés rencontrées et précautions à prendre lors de la programmation ou de l'utilisation d'algorithmes de calcul numérique dans les disciplines qui y recourent.

Notions	**Commentaires**
Représentation des entiers positifs sur des mots de taille fixe.	La conversion d'une base à une autre n'est pas un objectif de formation.
Représentation des entiers signés sur des mots de taille fixe.	Complément à deux.

Entiers multi-précision de Python.	On les distingue des entiers de taille fixe sans détailler leur implémentation. On signale la difficulté à évaluer la complexité des opérations arithmétiques sur ces entiers.
Distinction entre nombres réels, décimaux et flottants.	On montre sur des exemples l'impossibilité de représenter certains nombres réels ou décimaux dans un mot machine
Représentation des flottants sur des mots de taille fixe. Notion de mantisse, d'exposant.	On signale la représentation de 0 mais on n'évoque pas les nombres dénormalisés, les infinis ni les NaN. Aucune connaissance liée à la norme IEEE-754 n'est au programme.
Précision des calculs en flottants.	On insiste sur les limites de précision dans le calcul avec des flottants, en particulier pour les comparaisons. Le comparatif des différents modes d'arrondi n'est pas au programme.

2.3 Bases des graphes, plus courts chemins

Il s'agit de définir le modèle des graphes, leurs représentations et leurs manipulations.
On s'efforce de mettre en avant des applications importantes et si possible modernes : réseau de transport, graphe du web, réseaux sociaux, bio-informatique. On précise autant que possible la taille typique de tels graphes.

Notions	**Commentaires**
Vocabulaire des graphes.	Graphe orienté, graphe non orienté. Sommet (ou nœud) ; arc, arête. Boucle. Degré (entrant et sortant). Chemin d'un sommet à un autre. Cycle. Connexité dans les graphes non orientés. On présente l'implémentation des graphes à l'aide de listes d'adjacence (rassemblées par exemple dans une liste ou dans un dictionnaire) et de matrice d'adjacence. On n'évoque ni multi-arcs ni multi-arêtes.
Notations.	Graphe $G = (S, A)$, degrés $d(s)$ (pour un graphe non orienté), $d_+(s)$ et $d_-(s)$ (pour un graphe orienté).
Pondération d'un graphe. Étiquettes des arcs ou des arêtes d'un graphe.	On motive l'ajout d'information à un graphe par des exemples concrets.
Parcours d'un graphe.	On introduit à cette occasion les piles et les files ; on souligne les problèmes d'efficacité posés par l'implémentation des files par les listes de Python et l'avantage d'utiliser un module dédié tel que `collections.deque`. Détection de la présence de cycles ou de la connexité d'un graphe non orienté.
Recherche d'un plus court chemin dans un graphe pondéré avec des poids positifs.	Algorithme de Dijkstra. On peut se contenter d'un modèle de file de priorité naïf pour extraire l'élément minimum d'une collection. Sur des exemples, on s'appuie sur l'algorithme A* vu comme variante de celui de Dijkstra pour une première sensibilisation à la notion d'heuristique.

3 Programme du troisième semestre

3.1 Bases de données

On se limite volontairement à une description applicative des bases de données en langage SQL. Il s'agit de permettre d'interroger une base présentant des données à travers plusieurs relations. On ne présente pas l'algèbre relationnelle ni le calcul relationnel.

Notions	**Commentaires**
Vocabulaire des bases de données : tables ou relations, attributs ou colonnes, domaine, schéma de tables, enregistrements ou lignes, types de données.	On présente ces concepts à travers de nombreux exemples. On s'en tient à une notion sommaire de domaine : entier, flottant, chaîne ; aucune considération quant aux types des moteurs SQL n'est au programme. Aucune notion relative à la représentation des dates n'est au programme ; en tant que de besoin on s'appuie sur des types numériques ou chaîne pour lesquels la relation d'ordre coïncide avec l'écoulement du temps. Toute notion relative aux collations est hors programme ; on se place dans l'hypothèse que la relation d'ordre correspond à l'ordre lexicographique usuel. `NULL` est hors programme.
Clé primaire.	Une clé primaire n'est pas forcément associée à un unique attribut même si c'est le cas le plus fréquent. La notion d'index est hors programme.
Entités et associations, clé étrangère.	On s'intéresse au modèle entité–association au travers de cas concrets d'associations $1-1, 1-*, *-*$. Séparation d'une association $*-*$ en deux associations $1-*$. L'utilisation de clés primaires et de clés étrangères permet de traduire en SQL les associations $1-1$ et $1-*$.
Requêtes `SELECT` avec simple clause `WHERE` (sélection), projection, renommage `AS`. Utilisation des mots-clés `DISTINCT`, `LIMIT`, `OFFSET`, `ORDER BY`.	Les opérateurs au programme sont `+`, `-`, `*`, `/` (on passe outre les subtilités liées à la division entière ou flottante), `=`, `<>`, `<`, `<=`, `>`, `>=`, `AND`, `OR`, `NOT`.
Opérateurs ensemblistes `UNION`, `INTERSECT` et `EXCEPT`, produit cartésien.	
Jointures internes T_1 `JOIN` T_2 … `JOIN` T_n `ON` ϕ. Autojointure.	On présente les jointures en lien avec la notion de relations entre tables. On se limite aux équi-jointures : ϕ est une conjonction d'égalités.
Agrégation avec les fonctions `MIN`, `MAX`, `SUM`, `AVG` et `COUNT`, y compris avec `GROUP BY`.	Pour la mise en œuvre des agrégats, on s'en tient à la norme SQL99. On présente quelques exemples de requêtes imbriquées.
Filtrage des agrégats avec `HAVING`.	On marque la différence entre `WHERE` et `HAVING` sur des exemples.

Mise en œuvre
La création de tables et la suppression de tables au travers du langage SQL sont hors programme. La mise en œuvre effective se fait au travers d'un logiciel permettant d'interroger une base de données à l'aide de requêtes SQL. Récupérer le résultat d'une requête à partir d'un programme n'est pas un objectif. Même si aucun formalisme graphique précis n'est au programme, on peut décrire les entités et les associations qui les lient au travers de diagrammes sagittaux informels. Sont hors programme : la notion de modèle logique *vs* physique, les bases de données non relationnelles, les méthodes de modélisation de base, les fragments DDL, TCL et ACL du langage SQL, les transactions, l'optimisation de requêtes par l'algèbre relationnelle.

3.2 Dictionnaires et programmation dynamique

Les dictionnaires sont utilisés en boîte noire dès la première année ; les principes de leur fonctionnement sont présentés en deuxième année. Ils peuvent être utilisés afin de mettre en mémoire des résultats intermédiaires quand on implémente une stratégie d'optimisation par programmation dynamique.

Notions	**Commentaires**
Dictionnaires, clés et valeurs.	On présente les principes du hachage, et les limitations qui en découlent sur le domaine des clés utilisables.
Usage des dictionnaires en programmation Python.	Syntaxe pour l'écriture des dictionnaires. Parcours d'un dictionnaire.
Programmation dynamique. Propriété de sous-structure optimale. Chevauchement de sous-problèmes. Calcul de bas en haut ou par mémoïsation. Reconstruction d'une solution optimale à partir de l'information calculée.	La mémoïsation peut être implémentée à l'aide d'un dictionnaire. On souligne les enjeux de complexité en mémoire. Exemples : partition équilibrée d'un tableau d'entiers positifs, ordonnancement de tâches pondérées, plus longue sous-suite commune, distance d'édition (Levenshtein), distances dans un graphe (Floyd-Warshall).
Mise en œuvre	
Les exemples proposés ne forment une liste ni limitative ni impérative. Les cas les plus complexes de situations où la programmation dynamique peut être utilisée sont guidés. On met en rapport le statut de la propriété de sous-structure optimale en programmation dynamique avec sa situation en stratégie gloutonne vue en première année.	

3.3 Algorithmique pour l'intelligence artificielle et l'étude des jeux

Cette partie permet notamment de revisiter les notions de programmation et de représentation de données par un graphe, qui sont vues en première année, en les appliquant à des enjeux contemporains.

Notions	**Commentaires**
Algorithme des k plus proches voisins avec distance euclidienne.	Matrice de confusion. Lien avec l'apprentissage supervisé.
Algorithme des k-moyennes.	Lien avec l'apprentissage non-supervisé. La démonstration de la convergence n'est pas au programme. On observe des convergences vers des minima locaux.
Jeux d'accessibilité à deux joueurs sur un graphe. Stratégie. Stratégie gagnante. Position gagnante. Détermination des positions gagnantes par le calcul des attracteurs. Construction de stratégies gagnantes.	On considère des jeux à deux joueurs (J_1 et J_2) modélisés par des graphes bipartis (l'ensemble des états contrôlés par J_1 et l'ensemble des états contrôlés par J_2). Il y a trois types d'états finals : les états gagnants pour J_1, les états gagnants pour J_2 et les états de match nul. On ne considère que les stratégies sans mémoire.
Notion d'heuristique. Algorithme min-max avec une heuristique.	L'élagage alpha-beta n'est pas au programme.
Mise en œuvre	
La connaissance dans le détail des algorithmes de cette section n'est pas un attendu du programme. Les étudiants acquièrent une familiarité avec les idées sous-jacentes qu'ils peuvent réinvestir dans des situations où les modélisations et les recommandations d'implémentation sont guidées, notamment dans leurs aspects arborescents.	

A Langage Python

Cette annexe liste limitativement les éléments du langage Python (version 3 ou supérieure) dont la connaissance est exigible des étudiants. Aucun concept sous-jacent n'est exigible au titre de la présente annexe.
Aucune connaissance sur un module particulier n'est exigible des étudiants.
Toute utilisation d'autres éléments du langage que ceux que liste cette annexe, ou d'une fonction d'un module, doit obligatoirement être accompagnée de la documentation utile, sans que puisse être attendue une quelconque maîtrise par les étudiants de ces éléments.

Traits généraux

- Typage dynamique : l'interpréteur détermine le type à la volée lors de l'exécution du code.
- Principe d'indentation.
- Portée lexicale : lorsqu'une expression fait référence à une variable à l'intérieur d'une fonction, Python cherche la valeur définie à l'intérieur de la fonction et à défaut la valeur dans l'espace global du module.
- Appel de fonction par valeur : l'exécution de $f(x)$ évalue d'abord x puis exécute f avec la valeur calculée.

Types de base

- Opérations sur les entiers (`int`) : `+`, `-`, `*`, `//`, `**`, `%` avec des opérandes positifs.
- Opérations sur les flottants (`float`) : `+`, `-`, `*`, `/`, `**`.
- Opérations sur les booléens (`bool`) : `not`, `or`, `and` (et leur caractère paresseux).
- Comparaisons `==`, `!=`, `<`, `>`, `<=`, `>=`.

Types structurés

- Structures indicées immuables (chaînes, tuples) : `len`, accès par indice positif valide, concaténation `+`, répétition `*`, tranche.
- Listes : création par compréhension `[`e `for` x `in` s`]`, par `[`e`] *` n, par `append` successifs ; `len`, accès par indice positif valide ; concaténation `+`, extraction de tranche, copie (y compris son caractère superficiel) ; `pop` en dernière position.
- Dictionnaires : création `{`c_1 `:` v_1`, …,` c_n `:` v_n`}`, accès, insertion, présence d'une clé k `in` d, `len`, `copy`.

Structures de contrôle

- Instruction d'affectation avec `=`. Dépaquetage de tuples.
- Instruction conditionnelle : `if`, `elif`, `else`.
- Boucle `while` (sans `else`). `break`, `return` dans un corps de boucle.
- Boucle `for` (sans `else`) et itération sur `range(`a`,` b`)`, une chaîne, un tuple, une liste, un dictionnaire au travers des méthodes `keys` et `items`.
- Définition d'une fonction `def` f`(`p_1`, …,` p_n`)`, `return`.

Divers

- Introduction d'un commentaire avec `#`.
- Utilisation simple de `print`, sans paramètre facultatif.
- Importation de modules avec `import` *module*, `import` *module* `as` *alias*, `from` *module* `import` $f, g, \ldots$

- Manipulation de fichiers texte (la documentation utile de ces fonctions doit être rappelée ; tout problème relatif aux encodages est éludé) : `open`, `read`, `readline`, `readlines`, `split`, `write`, `close`.
- Assertion : `assert` (sans message d'erreur).