Livret de révision du programme de spécialité NSI de terminale

Dictionnaires

CE QU'IL FAUT SAVOIR

- Un **dictionnaire** est un type construit de **données**
- Chaque élément d'un dictionnaire est composé de deux parties ; on parle de paire **`clé: valeur`**
- Pour afficher une **valeur particulière,** on utilise la notation **`mon_dico[nom_de_la_clé ]`**
- Il est possible de **parcourir l'ensemble des clés** d'un dictionnaire à l'aide d'une boucle for en utilisant la méthode **`.keys()`**
- Il est possible de **parcourir l'ensemble des valeurs** d'un dictionnaire à l'aide d'une boucle for en utilisant la méthode **`.values()`**
- Il est possible de **parcourir l'ensemble des clés et des valeurs** (en même temps) d'un dictionnaire à l'aide d'une boucle for en utilisant la méthode **`.items()`**

Implémentation des dictionnaires

Dans de nombreux langages de programmation, les dictionnaires sont implémentés à l'aide de **tables de hachage**.

Algorithme de recherche dans un dictionnaire

L'algorithme de recherche d'un élément dans un dictionnaire a une complexité $O(1)$. La durée de recherche ne dépend pas du nombre d'éléments présents dans le dictionnaire.

CE QU'IL FAUT SAVOIR FAIRE

- **Construire** un dictionnaire :
  - dictionnaire vide : **`d = dict()`** ou **`d =`** {} ;
  - **`mon_dico =`** {**`"nom": "Dupont", "prenom": "Jean", "date de naissance": "17/05/1971"`**}.
- **Utiliser la notation `mon_dico[nom_de_la_clé ]`** afin d'ajouter une nouvelle paire **`clé: valeur`**, d'utiliser une valeur déjà existante ou de modifier la valeur courante par une autre valeur (la clé restant identique)
- **Utiliser la méthode `get`** pour récupérer une valeur dans un dictionnaire : **`mon_dico.get("adresse", "Adresse introuvable")`**
- **Suprimer une clé** et la valeur qui lui est associée : **`del mon_dico["nom"]`**
- **Parcourir l'ensemble des clés**
- **Parcourir l'ensemble des valeurs**
- **Parcourir l'ensemble des clés et des valeurs** (en même temps)

Récursivité

CE QU'IL FAUT SAVOIR

- Une **fonction récursive** est une fonction qui s'appelle elle-même
- Quand on écrit une fonction récursive, il est nécessaire de bien penser à mettre en place une structure qui à un moment ou à un autre mettra fin à ces appels récursifs
- L'utilisation des fonctions récursives est souvent liée à la notion de **récurrence** en mathématiques

CE QU'IL FAUT SAVOIR FAIRE

- Analyser le fonctionnement d'un programme récursif (programme qui comporte au moins une fonction récursive)
- Être capable d'écrire une fonction récursive simple

Paradigmes de programmation

CE QU'IL FAUT SAVOIR

Paradigme fonctionnel

- La **programmation fonctionnelle** est un paradigme de programmation comme la programmation impérative ou la programmation orientée objet
- Le paradigme fonctionnel repose sur l'utilisation de **fonctions**
- Le paradigme fonctionnel cherche à éviter au maximum les **effets de bord** : on s'efforce de coder des fonctions qui ne modifient pas l'état courant des variables globales
- Les fonctions utilisées en programmation fonctionnelle sont parfois appelées **fonctions pures** ou encore **citoyennes de premère classe** : le résultat renvoyé par une telle fonction doit uniquement dépendre des paramètres passés à la fonction et pas des valeurs externes à la fonction

Paradigme objet

La **programmation orientée objet** (expression souvent abrégée en POO) est un paradigme de programmation qui repose sur la notion de **classe,** la notion d'**attribut** et la notion de **méthode** (la POO repose aussi sur les notions d'**héritage** et de **polymorphisme,** mais ces deux notions ne sont pas au programme de NSI).

En POO on travaille sur des **objets** (des **instances** plus exactement) : chaque instance est créée à partir d'un “moule” (la classe) ; les attributs représentent l'état d'un objet alors que les méthodes en représentent le comportement

CE QU'IL FAUT SAVOIR FAIRE

Paradigme fonctionnel

Être capable d'écrire un programme simple en Python qui respecte le paradigme fonctionnel

Paradigme objet

- Être capable d'analyser et comprendre un programme Python simple qui utilise le paradigme objet
- Être capable d'écrire un programme Python simple qui utilise le paradigme objet

Systèmes de gestion de base de données

CE QU'IL FAUT SAVOIR

Les bases de données permettent de stocker des données. Pour manipuler les données présentes dans une base de données (écrire, lire ou encore modifier), il est nécessaire d'utiliser un type de logiciel appelé **système de gestion de base de données** très souvent abrégé en **SGBD.**

- Les SGBD permettent de gérer la lecture, l'écriture ou la modification des informations contenues dans une base de données
- Les SGBD permettent de gérer les autorisations d'accès à une base de données
- Les SGBD assurent la maintenance des différentes copies de la base de données (en cas de panne d'un ordinateur) : on parle de redondance des données
- Les problèmes d'accès concurrent (plusieurs personnes connectées en même temps) sont gérés par les SGBD

Avantages d'un système de gestion de base de données

- Une gestion simple des grands ensembles de données
- Un accès simple et efficace aux données enregistrées
- Une grande flexibilité
- L'intégrité et la cohérence des données
- Le contrôle des accès pour les utilisateurs (sécurité et protection des données)
- Une disponibilité élevée

Inconvénients d'un système de gestion de base de données

- Un investissement de départ relativement plus coûteux (incluant les coûts supplémentaires pour le matériel)
- Plutôt moins efficace pour les logiciels spéciaux
- Nécessite des employés qualifiés (administrateurs de bases de données)
- Une vulnérabilité importante du fait de la centralisation des données

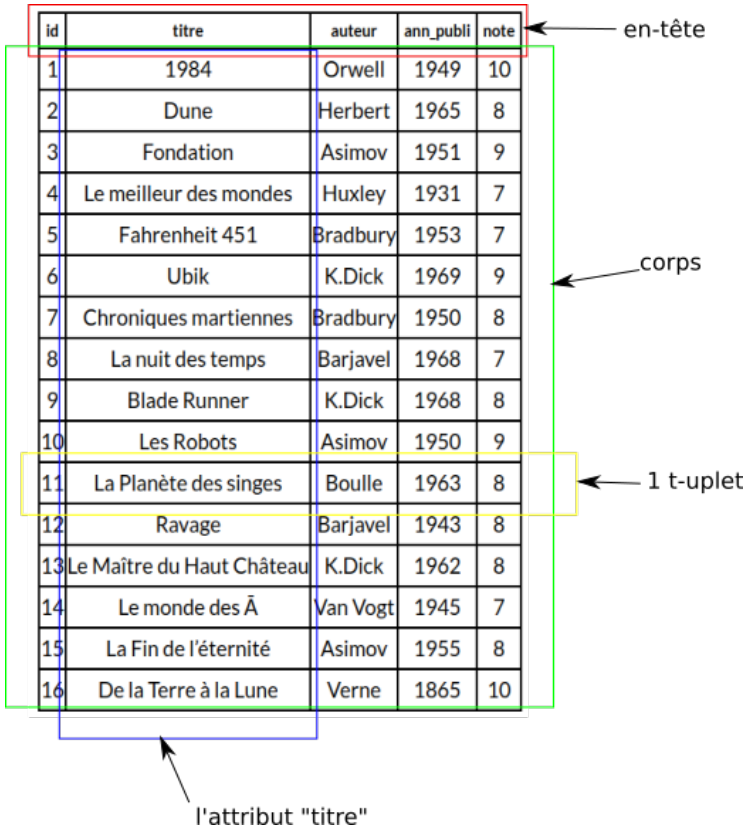
Exemples de SGBD

- Mysql – relationnel
- PostgreSQL – relationnel et objet (mélange des approches relationnelle et objet)
- MariaDB – relationnel
- Oracle Database – relationnel
- MongoDB – orienté documents
- Microsoft Access – relationnel

Bases de données relationnelle

CE QU'IL FAUT SAVOIR

- Il existe des bases de données relationnelles, hiérarchiques, orientées réseau, orientées objet, etc.
- Le modèle de **base de données relationnel** est le plus courant et le plus apprécié ; la notion de **relation** est au coeur des bases de données relationnelles ; une relation peut être vue comme un **tableau à deux dimensions**, composé d'un **en-tête** et d'un **corps**; le corps est lui-même composé de **t-uplets** (lignes) et d'**attributs** (colonnes) ; l'en-tête contient les intitulés des attributs, le corps contient les données proprement dites



id	titre	auteur	ann_publi	note
1	1984	Orwell	1949	10
2	Dune	Herbert	1965	8
3	Fondation	Asimov	1951	9
4	Le meilleur des mondes	Huxley	1931	7
5	Fahrenheit 451	Bradbury	1953	7
6	Ubik	K.Dick	1969	9
7	Chroniques martiennes	Bradbury	1950	8
8	La nuit des temps	Barjavel	1968	7
9	Blade Runner	K.Dick	1968	8
10	Les Robots	Asimov	1950	9
11	La Planète des singes	Boulle	1963	8
12	Ravage	Barjavel	1943	8
13	Le Maître du Haut Château	K.Dick	1962	8
14	Le monde des Ã	Van Vogt	1945	7
15	La Fin de l'éternité	Asimov	1955	8
16	De la Terre à la Lune	Verne	1865	10

- Pour chaque attribut d'une relation, il est nécessaire de définir un **domaine** : le domaine d'un attribut donné correspond à un ensemble fini ou infini de valeurs admissibles
- Une **clé primaire** est un attribut dont la valeur permet d'identifier de **manière unique** un t-uplet de la relation
- Une **clé étrangère** est un attribut d'une relation A devant apparaître comme clé primaire dans une relation B afin d'établir un lien entre A et B (une clé étrangère d'un tuple référence une clé primaire d'un autre tuple) ; la notion de clé étrangère permet de préserver l'**intégrité** d'une base de données lorsque l'on travail sur plusieurs relations en même temps
- On appelle **schéma relationnel** l'ensemble des relations présentes dans une base de données ; dans le schéma relationnel on doit trouver :
  - les noms des différentes relations ;
  - pour chaque relation, la liste des attributs avec leur domaine respectif ;
  - pour chaque relation, la clé primaire (**soulignée**) et éventuellement les clés étrangères (**précédées d'un #**).

Exemples

`Auteurs(_id_ : INT, nom : TEXT, prenom : TEXT, ann_naissance : INT, langue_ecriture : TEXT)`

`Livres(_id_ : INT, titre : TEXT, #id_auteur : INT, ann_publi : INT, note : INT)`

Langage SQL

CE QU'IL FAUT SAVOIR

- Pour consulter des données, ajouter une entrée, modifier une entrée ou supprimer une entrée dans une base de données relationnelle, il est nécessaire d'effectuer des requêtes **SQL** (utilisation du langage SQL)
- Pour **ajouter des entrées à une table**, on utilisera **`INSERT`** :
  - **`INSERT INTO Livres (id, titre, auteur, ann_publi, note) VALUES (1, '1984', 'Orwell', 1949, 10);`**
- Pour **interroger une table**, on utilisera **`SELECT`** :
  - **`SELECT titre FROM Livres WHERE auteur = 'Asimov' ;`**
- Pour **modifier une entrée**, on utilisera **`UPDATE`** :
  - **`UPDATE Livres SET note = 7 WHERE titre = 'Hypérion' ;`**
- Pour **supprimer une entrée**, on utilisera **`DELETE`** :
  - **`DELETE FROM Livres WHERE titre = 'Hypérion' ;`**
- Pour **réaliser une jointure**, il est possible d'utiliser **`INNER JOIN`** :
  - **`SELECT FROM Livres INNER JOIN Auteurs ON Livres.id_auteur = Auteurs.id ;`**

CE QU'IL FAUT SAVOIR FAIRE

- Vous devez être capable d'effectuer des requêtes SQL simples (utilisation de **`INSERT`**, **`SELECT`**, **`UPDATE`** et **`DELETE`**)
- Vous devez être capable d'effectuer une jointure entre deux tables (utilisation de **`INNER JOIN`**)

Mise au point des programmes et débogage

CONTEXTE

Programmer sans erreur est une tâche difficile en raison de la taille des logiciels, du nombre de personnes impliquées dans leur confection et de leur historique. Pourtant l'enjeu est de taille : une erreur dans un programme peut conduire à un blocage, à une perte financière… voire à la mort.

Quelques exemples de conséquences désastreuses de programmes mal conçus :

- 1996 : destruction de la fusée Ariane 5 à cause du mauvais typage d'une variable ;
- 1985 : le logiciel *Therac-25* a provoqué le décès de plusieurs patients à cause d'erreurs de conception et de l'absence de tests ;
- 2004 : blocage de la billeterie aux guichets de la SNCF pendant deux jours dû à un problème de mise à jour d'un logiciel.

MISE AU POINT

Pour mettre au point un programme dans de bonnes conditions, il y a de **bonnes habitudes** à developper :

- bien **analyser** le problème à résoudre ;
- écrire du **code simple** et être à l'affût de simplifications ;
- écrire du **code lisible** et **compréhensible** ;
- utiliser des **assertions** (`assert`) pour contrôler :
  - les **préconditions** vérifiées par les variables avant leur traitement par une fonction, par exemple qu'une variable est bien supérieure à une autre, qu'une liste ou une chaine de caractères n'est pas vide, qu'une variable est bien du bon type, etc.
  - les **postconditions** vérifiées avant le `return` pour s'assurer que le traitement est conforme aux attentes, le fait qu'une fonction renvoie un résultat du bon type, positif, non vide, etc.
- écrire des **tests unitaires** (`doctest`, `unittest`, `pytest`, etc.) : "les tests peuvent servir à montrer la présence d'erreurs, pas leur absence", E. Dijkstra ;
- corriger les **bugs** avant d'ajouter de nouvelles fonctionnalités ;
- tout **documenter** (**docstring)** et **commenter** ;
- suivre régulièrement les **technologies** utilisées et leurs alternatives.

DÉBOGAGE

Les environnements de développement modernes ont tous des **débogueurs** intégrés. Ces derniers permettent de définir des **points d'arrêt**, d'observer les valeurs des variables (en passant la souris dessus), d'afficher la **pile d'appels** et d'exécuter le programme **pas à pas** (il existe des icones spécifiques).

Parmi les **causes typiques** de bugs, on retrouve :

- les problèmes liés au **typage** ;
- les **effets de bord** non désirés ;
- les **débordements** dans les tableaux (listes en Python) ;
- les instructions conditionnelles **non exhaustives** ;
- les choix des inégalités et des **opérateurs booléens** ;
- les comparaisons et calculs entre **flottants** ;
- un **mauvais nommage** des variables.

Listes, piles et files

CE QU'IL FAUT SAVOIR

Les listes

Une **liste** est une structure de données permettant de regrouper des données. Une liste `L` est composée de deux parties : sa **tête** (**head** en anglais), qui correspond au dernier élément ajouté à la liste, et sa **queue** (`tail` en anglais) qui correspond au reste de la liste. Voici les opérations qui peuvent être effectuées sur une liste :

- création d'une liste vide ;
- ajout d'un élément en tête / en queue / en position *i* ;
- accès à l'élément en tête / en queue / au *i*-ième élément ;
- modification d'un élément en tête / en queue / du *i*-ième élément ;
- suppression en tête / en queue / en position *i* ;
- longueur ;
- concaténation de deux listes.

Une implémentation existante de ce type abstrait est le type prédéfini `list` de Python qui utilise des tableaux dynamiques. On peut aussi réimplanter de manière élémentaire le type **liste** par des **listes simplement chaînées** en utilisant la programmation objet. Pour cela, on doit choisir de définir un ensemble de méthodes élémentaires, et redéfinir les autres en fonction des méthodes élémentaires choisies. On choisit une **représentation non contiguë des listes**, avec des **cellules comportant chacune un élément et une référence au suivant**. C'est donc une structure de données récursive.

Les piles

On retrouve dans les piles une partie des propriétés vues sur les listes. Dans les piles, il est uniquement possible de manipuler le dernier élément introduit dans la pile. Les piles sont basées sur le principe **LIFO**. Voici les opérations que l'on peut réaliser sur une pile :

- créer une pile vide ;
- savoir si une pile est vide ;
- empiler un nouvel élément sur la pile (**push** en anglais) ;
- récupérer l'élément au sommet de la pile tout en le supprimant ; on dit que l'on dépile (**pop** en anglais) ;
- connaitre le nombre d'éléments présents dans la pile.

Les files

Comme les piles, les files ont des points communs avec les listes. Différences majeures : dans une file on ajoute des éléments à une extrémité de la file et on supprime des éléments à l'autre extrémité. Les files sont basées sur le principe **FIFO.** Voici les opérations que l'on peut réaliser sur une file :

- créer une file vide ;
- savoir si une file est vide ;
- ajouter un nouvel élément à la file (enfiler en français, **enqueue** en anglais) ;
- récupérer l'élément situé en bout de file tout en le supprimant (défiler en français, **dequeue** en anglais) ;
- connaitre le nombre d'éléments présents dans la file.

CE QU'IL FAUT SAVOIR FAIRE

Implémenter les structures abstraites `liste`, `pile` et `file` en Python

Les arbres

CE QU'IL FAUT SAVOIR

Généralités

Un arbre est un **graphe sans cycle**, où des nœuds sont reliés par des **arêtes.** C'est un type abstrait très utilisé quand on a besoin d'une structure hiérarchique des données. On distingue trois sortes de nœuds :

- les **nœuds internes**, qui ont des fils ;
- les **feuilles,** qui n'ont pas de fils ;
- la **racine** de l'arbre, qui est l'unique nœud ne possédant pas de père.

Traditionnellement, on dessine toujours la racine en haut et les feuilles en bas.

- Le **degré** d'un nœud est le nombre de fils de ce nœud
- La **profondeur** d'un nœud est la distance, c'est-à-dire le nombre d'arêtes, de la racine au nœud
- La **hauteur** d'un arbre est la plus grande profondeur d'une feuille de l'arbre
- La **taille** d'un arbre est son nombre de nœuds

Les arbres peuvent être étiquetés. Dans ce cas, chaque nœud possède une **étiquette (ou clé)** qui est en quelque sorte le « contenu » du nœud. L'étiquette peut être très simple (un nombre entier, par exemple) ou plus complexe (un objet, une instance d'une structure de données, etc).

On peut représenter un arbre sous forme :

- d'une **liste d'adjacence** (un *dictionnaire* en Python) ;
- d'un **tableau à deux dimensions** (une *liste de listes* en Python).

On peut voir les arbres comme des structures récursives : les fils d'un nœud sont des arbres (sous-arbre gauche et un sous-arbre droite dans le cas d'un arbre binaire), ces arbres sont eux-mêmes constitués d'arbres, et ainsi de suite.

Arbre binaire

Dans un **arbre binaire,** chaque nœud possède au plus deux fils, habituellement appelés « gauche » et « droit ». Du point de vue des fils, l'élément dont ils sont issus au niveau supérieur est logiquement appelé père.

Pour un **arbre binaire quelconque**, on a :

$$\lfloor log_2(n) \rfloor \leq h \leq n-1$$

avec n la taille de l'arbre et h la hauteur de l'arbre ($\lfloor log_2(n) \rfloor$ étant la partie entière du logarithme base 2 de n).

Arbre binaire de recherche

Un **arbre binaire de recherche** est un arbre binaire tel que :

- les clés des noeuds composant l'arbre sont ordonnables (on doit pouvoir classer les noeuds, par exemple, de la plus petite clé à la plus grande) ;
- soit x un nœud d'un arbre binaire de recherche :
  - si y est un nœud du sous-arbre gauche de x, alors il faut que **`y.clé <= x.clé`** ;
  - si y est un nœud du sous-arbre droit de x, il faut alors que **`x.clé <= y.clé`**.

Pour un nœud donné A, tous les noeuds de l'arbre gauche de A auront des valeurs plus petites que la valeur du nœud A et tous les noeuds de la l'arbre droit de A auront des valeurs plus grandes que la valeur du nœud A.

Algorithmes sur les arbres binaires

CE QU'IL FAUT SAVOIR

- Connaître l'algorithme qui permet de **calculer la hauteur d'un arbre**
- Connaître l'algorithme qui permet de **calculer la taille d'un arbre**
- Connaître les algorithmes qui permettent de **parcourir** un arbre : **ordre infixe, ordre préfixe, ordre suffixe, en largeur d'abord**
- Connaître l'algorithme qui permet de **rechercher une clé (étiquette) dans un arbre binaire de recherche**, savoir que cet algorithme à une complexité en $O\left(log_2(n)\right)$ dans le cas d'un arbre binaire de recherche équilibré et $O(n)$ dans le cas d'un arbre binaire de recherche filiforme.
- Connaître l'algorithme qui permet d'**insérer une clé dans un arbre binaire de recherche**

CE QU'IL FAUT SAVOIR FAIRE

Être capable d'implémenter tous ces algorithmes en Python

Diviser pour régner

CE QU'IL FAUT SAVOIR

Diviser pour régner (*divide and conquer* en anglais) est une méthode algorithmique basée sur le principe suivant : on prend un problème (généralement complexe à résoudre), on divise ce problème en une multitude de petits problèmes, l'idée étant que ceux-ci seront plus simples à résoudre que le problème original. Une fois les petits problèmes résolus, on les recombine afin d'obtenir la solution du problème de départ.

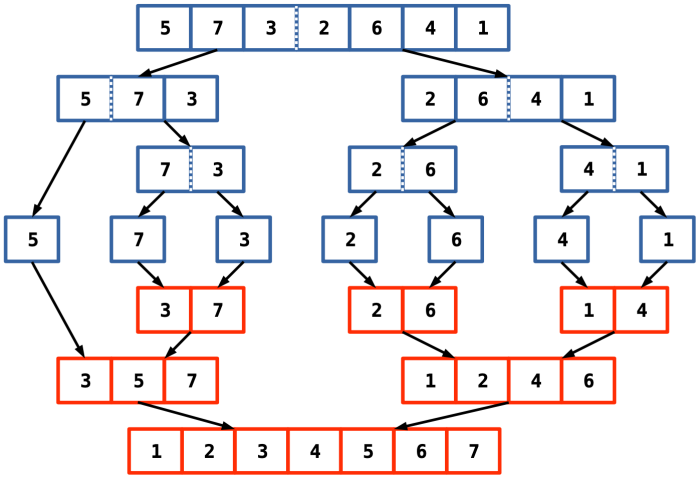
Le paradigme *diviser pour régner* repose donc sur trois étapes :

- **diviser** : le problème d'origine est divisé en un certain nombre de sous-problèmes ;
- **régner** : on résout les sous-problèmes (qui sont plus faciles à résoudre que le problème d'origine) ;
- **combiner** : les solutions des sous-problèmes sont combinées afin d'obtenir la solution du problème d'origine.

Ce paradigme tire avantage de la **récursivité.**

Exemple d'algorithme basé sur la méthode diviser pour régner

L'algorithme de tri fusion s'appuie sur cette méthode pour trier un tableau efficacement (une liste en Python), avec une complexité en $O(n \log_2 n)$ dans le pire cas. Il consiste à scinder en deux une liste à trier, puis à trier ces deux sous-listes (de manière récursive jusqu'à obtenir des listes de longueur 1) et enfin à fusionner ces deux sous-listes triées.



CE QU'IL FAUT SAVOIR FAIRE

Être capable d'implémenter l'algorithme de tri par fusion en Python

Programmation dynamique

CE QU'IL FAUT SAVOIR

- La **programmation dynamique** est une méthode algorithmique utilisée pour résoudre les problèmes **d'optimisation** (comme les méthodes gloutonnes vues en classe de première)
- La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands **en stockant les résultats intermédiaires**

CE QU'IL FAUT SAVOIR FAIRE

Être capable d'utiliser la programmation dynamique dans des cas simples (comme, par exemple, le problème du rendu de monnaie ou celui du sac à dos)

Les graphes

CE QU'IL FAUT SAVOIR

Généralités

- Un **graphe** est un objet mathématique très utilisé en informatique
- Un graphe possède des **sommets** et des **arêtes** : un graphe G est un couple $G = (V, E)$ avec V un ensemble de sommets et E un ensemble d'arêtes
- Les graphes sont, par exemple, utilisés pour modéliser les réseaux sociaux ou encore les cartes routières
- On trouve deux types de graphes : les **graphes non orientés** et les **graphes orientés**. Dans le cas des graphes orientés on parle d'**arcs** plutôt que d'arêtes
- On trouve des graphes dits **pondérés** : à chaque arête (ou arc) on associe une valeur
- Une **chaîne** est une suite d'arêtes consécutives dans un graphe, un peu comme si on se promenait sur le graphe ; on la désigne par les lettres des sommets qu'elle comporte
- Un **cycle** est une chaîne qui commence et se termine au même sommet

Implémentations des graphes

- Il existe deux méthodes permettant d'implémenter un graphe : les **matrices d'adjacences** et les **listes d'adjacences**
- Le choix se fait en fonction de la densité du graphe, c'est-à-dire du rapport entre le nombre d'arêtes et le nombre de sommets ; pour un graphe dense on utilisera plutôt une matrice d'adjacence
- Certains algorithmes travaillent plutôt avec les listes d'adjacences alors que d'autres travaillent plutôt avec les matrices d'adjacences ; le choix dépend donc aussi des algorithmes utilisés

CE QU'IL FAUT SAVOIR FAIRE

- Être capable de déterminer la matrice d'adjacence d'un graphe à partir d'un schéma (et vice versa)
- Être capable de déterminer la liste d'adjacence d'un graphe à partir d'un schéma (et vice versa)

Algorithmes sur les graphes

CE QU'IL FAUT SAVOIR

- Connaître l'algorithme qui permet de **parcourir** un graphe en **largeur d'abord**
- Connaître l'algorithme qui permet de **parcourir** un graphe en **profondeur d'abord**
- Connaître l'algorithme qui permet de **détecter** les **cycles** dans un graphe
- Connaître l'algorithme qui permet de **chercher** une **chaîne** dans un graphe

CE QU'IL FAUT SAVOIR FAIRE

Être capable d'implémenter tous ces algorithmes en Python

Recherche textuelle

CE QU'IL FAUT SAVOIR

- Les algorithmes permettant de rechercher un motif (suite de lettres) dans un texte ont une grande importance en informatique
- L'**algorithme de Boyer-Moore** effectue un **prétraitement du motif**. Cela signifie que l'algorithme "connait" les caractères qui se trouvent dans le motif
- Dans l'algorithme de Boyer-Moore on commence la comparaison motif-chaine par la droite du motif. Par exemple pour le motif **CGGCAG**, on compare d'abord le **G**, puis le **A**, puis **C**, etc. On parcourt le motif de la droite vers la gauche
- Dans le cas de l'**algorithme naïf**, les décalages du motif vers la droite se faisaient toujours d'un cran à la fois.
- L'intérêt de l'algorithme de Boyer-Moore, c'est qu'il permet, dans certaines situations, d'effectuer un décalage de plusieurs crans en une seule fois
- Plus le motif est grand et plus l'algorithme de Boyer-Moore sera **efficace** par rapport à l'algorithme naïf

CE QU'IL FAUT SAVOIR FAIRE

Être capable d'appliquer l'algorithme de Boyer-Moore sur un cas simple

Sécurisation des communications

Lorsque deux personnes communiquent par l'intermédiaire d'un réseau informatique, elles sont en droit d'exiger que cette communication se déroule de façon **sécurisée,** autrement dit qu'elle respecte les principes de **confidentialité** (les deux personnes doivent être les seuls à lire leurs messages), d'**intégrité** (tous les messages dans leur intégralité ont bien été envoyés et réceptionnés) et d'**authentification** (chaque personne doit être sûre de bien échanger avec la bonne personne).

CE QU'IL FAUT SAVOIR

- Connaître le principe du **chiffrement symétrique** : utilisation d'une clé de chiffrement privé partagée par les deux interlocuteurs
- Connaître le principe du **chiffrement asymétrique** : utilisation d'une **clé publique** et d'une **clé privée**. Seule la clé publique est diffusée. A chiffre un message à l'aide de la clé publique de B puis envoie ce message chiffré à B ; B utilise sa clé privée pour déchiffrer le message envoyé par A
- Connaître le principe du **protocole HTTPS** : utilisation du chiffrement asymétrique pour partager une clé K puis utilisation du chiffrement symétrique (utilisation de la clé K) afin de chiffrer les données qui transitent sur le réseau

CE QU'IL FAUT SAVOIR FAIRE

Dans le cadre du chiffrement symétrique, être capable de chiffrer et déchiffrer un message à l'aide d'une clé

Protocoles de routage

CE QU'IL FAUT SAVOIR

- Un **routeur** permet de relier ensemble **plusieurs réseaux locaux**
- Chaque routeur possède une **table de routage**. Une table de routage peut être vue comme un tableau qui va contenir des informations permettant au routeur d'envoyer le paquet de données dans la "bonne direction"
- Il existe deux méthodes permettant de renseigner la table de routage d'un routeur :
  - le **routage statique** : chaque ligne doit être renseignée "à la main" ; cette solution est seulement envisageable pour des très petits réseaux de réseaux ;
  - le **routage dynamique** : tout se fait "automatiquement", on utilise des protocoles qui vont permettre de "découvrir" les différentes routes automatiquement afin de pouvoir remplir la table de routage tout aussi automatiquement.
- Un **réseau de réseaux** comportant des routeurs peut être modélisé par un graphe : chaque routeur est un sommet et chaque liaison entre les routeurs ou entre un routeur et un switch est une arête.
- Les algorithmes utilisés par les protocoles de routages sont donc des algorithmes issus de la théorie de graphes
- Les deux protocoles au programme de Terminale NSI sont les protocoles **RIP** (*Routing Information Protocol*) et **OSPF** (*Open Shortest Path First*) :
  - protocole RIP : le protocole RIP s'appuie sur l'algorithme de Bellman-Ford (algorithme qui permet de calculer les plus courts chemins dans un graphe) ; **il utilise le nombre de sauts comme métrique** ; ce protocole est aujourd'hui très rarement utilisé dans les grandes infrastructures ;
  - protocole OSPF : le protocole OSPF s'appuie sur l'algorithme de Dijkstra ; **il utilise le coût comme métrique** (la notion de coût est directement liée au débit des liaisons entre les routeurs).

CE QU'IL FAUT SAVOIR FAIRE

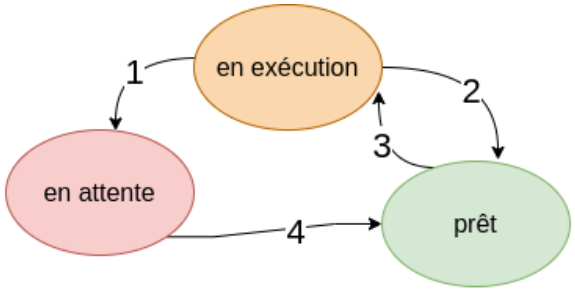
Être capable d'identifier la route empruntée par un paquet suivant le protocole de routage utilisé (RIP ou OSPF)

Gestion des processus

CE QU'IL FAUT SAVOIR

- On appelle **processus** un programme en cours d'exécution
- Deux processus P1 et P2 peuvent être amenés à partager une même ressource (fichiers…)
- Tous les **systèmes d'exploitation modernes** (Linux, Windows, macOS, Android, iOS…) sont capables de gérer l'exécution de plusieurs processus en même temps. Mais pour être précis, cela n'est pas en véritable "en même temps", mais plutôt un "chacun son tour". Pour gérer ce "chacun son tour", les systèmes d'exploitation attribuent des **états** au processus
- Un processus peut donc se trouver dans différents états :
  - **prêt** (*ready):* le processus attend son tour pour prendre la main ;
  - **en exécution** (*running):* le processus a accès au processeur pour exécuter ses instructions ;
  - **en attente** (*sleeping)* : le processus attend qu'un événement se produise (saisie clavier, réception d'une donnée par le réseau ou le disque dur …) ;
  - **arrêté** (*stopped)* : le processus a fini son travail ou a reçu un signal de terminaison (SIGTERM, SIGKILL, …). Il libère les ressources qu'il occupe ;
  - **zombie** : une fois arrêté, le processus informe son parent afin que ce dernier l'élimine de la table des processus ; cet état est donc temporaire mais il peut durer si le parent meurt avant de pouvoir effectuer cette tâche ; dans ce cas, le processus fils reste à l'état zombie.

Les trois premiers états sont les plus importants puisqu'ils décrivent le cycle de vie normal d'un processus :



1 : Le processus se met en attente d'un événement
2 : L'ordonnanceur passe la main à un autre processus
3 : L'ordonnanceur choisit ce processus
4 : L'événement attendu se produit

- Le système d'exploitation gère l'**ordonnancement des processus** (priorité d'exécution…)
- Chaque processus est créé par un autre processus (sauf le tout premier qui est créé au démarrage du système d'exploitation). Si un processus P crée un processus P', on dira que P est le **père** de P'
- Chaque processus possède un identifiant : le **PID** (*Process Identification*)
- Chaque processus possède aussi un **PPID** (*Parent Process Identification*) qui permet de connaître le processus parent d'un processus
- Dans certaines conditions, deux processus (ou plus) peuvent se trouver en situation **d'interblocage** (deadlock*)*

CE QU'IL FAUT SAVOIR FAIRE

Utiliser les commandes **Unix** qui permettent de :

- visualiser les processus en cours (**`ps -aef`**, **`top`**…) ;
- supprimer un processus (**`kill`**).

Systèmes sur puce

CE QU'IL FAUT SAVOIR

Définition

Il est aujourd'hui possible de faire tenir sur une puce d'une centaine de mm^2 l'ensemble des composants qui constitue un ordinateur classique (CPU, RAM, circuit graphique GPU :équivalent à la carte graphique dans un PC et circuits radio : Wifi et Bluetooth). Ces puces sont souvent appelées **système sur puce** en français, **system on a chip** en anglais (abréviation **SoC**).

Avantages

Outre leur taille, les Soc ont d'autres avantages par rapport aux systèmes classiques :

- les SoC sont conçus pour consommer beaucoup moins d'énergie qu'un système classique (à puissance de calcul équivalente) ;
- cette consommation réduite permet dans la plupart des cas de s'affranchir de la présence de système de refroidissement actif comme les ventilateurs ; un système équipé de SoC est donc silencieux ;
- vu les distances réduites entre, par exemple, le CPU et la mémoire, les données circulent beaucoup plus vites, ce qui permet d'améliorer les performances ; en effet, dans les systèmes classiques, les bus sont souvent des goulots d'étranglement en termes de performances à cause de la vitesse de circulation des données.

Exemples

On trouve des Soc dans :

- les smartphones ;
- les consoles de jeux portables (par exemple la switch de Nintendo) ;
- les nano-ordinateurs (par exemple le Raspberry Pi) ;
- et même des ordinateurs depuis 2021 (macbook et imac d'Apple).

Le marché des SoC est un marché extrêmement porteur.

Calculabilité - Décidabilité

CE QU'IL FAUT SAVOIR

- En 1937, **Alonzo Church** et **Alan Turing** ont démontré que certains problèmes ne pouvaient pas être résolus avec un algorithme
- S'il n'existe pas d'algorithme capable de résoudre un problème et que la réponse attendue à ce problème est “oui” ou “non”, on dira que ce problème est **indécidable** ; quand ce genre de problème peut être résolu à l'aide d'un algorithme, on dit que ce problème est **décidable**
- S'il n'existe pas d'algorithme capable de résoudre un problème et que la réponse attendue à ce problème est une valeur (issue d'un calcul), on dira que ce problème est **non-calculable**. Quand ce genre de problème peut être résolu à l'aide d'un algorithme, on dit que ce problème est **calculable**
- Si un problème est indécidable (ou non-calculable) cela ne veut pas dire que l'on n'est pas capable de résoudre ce problème, cela veut juste dire qu'il n'existe pas d'algorithme capable de résoudre ce problème
- Le **problème dit de l'arrêt** est indécidable

CE QU'IL FAUT SAVOIR FAIRE

Être capable de montrer que le problème dit de l'arrêt est indécidable

Intelligence artificielle

APPRENTISSAGE AUTOMATIQUE

L'**apprentissage automatique** (*machine learning* en anglais) est un domaine de l'**intelligence artificielle** qui utilise des techniques pour donner aux systèmes informatiques la possibilité "d'apprendre" (par exemple, d'améliorer progressivement les performances d'une tâche spécifique) à partir de données, sans être explicitement programmés.

En d'autre termes, c'est la science qui consiste à amener les ordinateurs à apprendre et à agir comme les humains, à améliorer leur apprentissage de manière autonome en leur fournissant des données et des informations sous forme d'observations et d'interactions dans le monde réel. L'apprentissage automatique est défini par une large gamme d'**algorithmes d'apprentissage**. Les algorithmes ne sont pas tous destinés aux mêmes usages. On les classe généralement selon deux catégories :

- le **mode d'apprentissage** : on distingue les algorithmes **supervisés** des algorithmes *non supervisés* ;
- le **type de problème à traiter** : on distingue les algorithmes de **régression** de ceux de **classification.**

Pour différencier les algorithmes d'apprentissage supervisé des algorithmes d'apprentissage non supervisé, prenons un exemple. Imaginons que l'on fournisse à l'algorithme des images représentant des chats et des chiens. S'il s'agit d'un algorithme supervisé, c'est alors nous, humains, qui allons proposer les critères de classement ("chat" ou "chien"). L'algorithme non supervisé doit par contre trouver lui–même les critères pour les organiser en groupe : on parle alors de **clustering.**

Concernant le type de problèmes que les algorithmes d'apprentissage peuvent traiter, la distinction classification/régression n'existe que dans le cas de l'apprentissage supervisé. L'exemple précédent est évidemment un algorithme de classification : c'est un chien ou un chat. On appelle cela une valeur discrète. Il n'y a pas de possibilité intermédiaire. À l'inverse, la régression permet d'obtenir un nombre réel. À partir d'un échantillon contenant les données d'habitation (quartier, superficie, etc.), l'algorithme sera en mesure de proposer un prix pour une nouvelle habitation.

ALGORITHME DES K PLUS PROCHES VOISINS

L'**algorithme des k plus proches voisins** est un **algorithme d'apprentissage supervisé**. Il est souvent noté k-NN pour *k-nearest neighbors* en anglais. Pour cet algorithme, nous devons disposer d'une **base de données d'apprentissage** qui servira à **entraîner notre algorithme**, qui pourra alors être utilisé sur des nouvelles données. La qualité de l'algorithme dépend donc des données présentes dans cette base de données. Trop peu nombreuses ou représentatives, l'algortihme ne pourra pas être utilisé efficacement.

Lorsqu'on exécute l'algorithme sur une nouvelle donnée, la méthode consiste à prendre en compte les k échantillons d'apprentissage les plus proches pour calculer le résultat. Dans l'exemple précédent, si **`k = 1000`** et que, parmi ces mille images, 986 sont des représentations de chiens (donc 14 sont des chats) alorsl'algorithme k-NN va considérer que la nouvelle image est un chien. Dans ce calcul, la notion de **distance entre échantillons** est primordiale et doit être définie en fonction des données et du fonctionnement attendu.

L'algorithme des k plus proches voisins est communément utilisé pour faire de la **classification** ou de la **régression.** En classification, l'algorithme est utilisé pour déterminer à quelle classe appartient la donnée. La classe résultat est typiquement la **classe majoritairement observée dans les k plus proches voisins**. Dans l'exemple, il y avait deux classes : soit "chien", soit "chat". Si k vaut 1, alors la donnée sera de la même classe que son plus proche voisin. En régression, le résultat n'est pas une classe mais un résultat calculé typiquement à partir de la moyenne des valeurs des voisins. C'était le cas de l'évaluation du prix d'une maison.