PARADIGMES DE PROGRAMMATION

Objet :
Mot clé `class` pour définir une classe.
Une classe comporte des *attributs* (équivalent des variables) et des *méthodes* (équivalent des fonctions).
Une classe est *instanciée* dans un objet du type de la classe.
Le *constructeur* de la classe est la méthode `__init__(self, paramètres)`, il crée l'objet et ses attributs. `self` est utilisé pour signifier que l'on fait appel à « soi-même », à l'instance de l'objet
On accède à un attribut de la classe par : `mon_instance.mon_attribut`
On utilise une méthode de la classe par : `mon_instance.ma_méthode(paramètres)` sans le self.

Fonctionnel :
Une fonction qui respecte le paradigme fonctionnel ne comporte pas de variables globales, n'a pas de variables modifiées par la fonction, et n'a pas de boucles (vision mathématique des fonctions). Il peut y avoir des « si » et de la récursivité.
Définition des tableaux dynamiques (type list de Python) ou des tableaux associatifs (type dict de Python) :

```
tab = [valeur for i in range(…)]  #valeur dépend presque toujours de i
dico = {cle : valeur for i in range(…)}  #clé (et/ou valeur) dépend toujours de i
```

Fonction `lambda` (ou anonyme), syntaxe : `lamba paramètre(s) : retour`
Une fonction lambda peut comporter un if (voire plusieurs if imbriqués) sous la forme

```
lambda paramètre(s) : retour1 if condition else retour2
```

RÉCURSIVITÉ

Une fonction récursive est une fonction qui s'appelle elle-même.
Pour qu'elle soit fonctionnelle, elle doit :

- comporter une condition d'arrêt (dans un cas simple en général).
- lorsqu'elle s'appelle, « diminuer » la taille des paramètres pour se rapprocher de la condition d'arrêt.

Les fonctions récursives utilisent une « pile d'appel » pour stocker les résultats intermédiaires.

Savoir dessiner un arbre ou une pile d'appels récursifs.

Structures linéaires de données.

Piles : ou LIFO (Last In First Out, dernier rentré premier sorti)
Files : ou FIFO (First In First Out, premier rentré premier sorti)

Définition avec des primitives (interface)/traduction sous forme d'objet :

Piles		
Interface / primitives	**Description**	**Implémentation en objet (fonctions ou méthodes)**
creer_pile() -> Pile	Créer une pile vide	`ma_pile = Pile()`
est_pile_vide(p) -> Booléen	Teste si la pile `p` est vide	`mon_booléen = ma_pile.est_pile_vide()`
empiler(`p` , `élément`)	Insère `élément` en tête de `p`	`ma_pile.empiler(élément)`
depiler(`p`) → `élément`	Enlève l'`élément` au sommet de la pile `p` et le renvoie	`élément_sommet = ma_pile.dépiler()`
sommet(`p`) -> `élément`	Renvoie l'`élément` au sommet de la pile `p`	`élément_sommet = ma_pile.sommet()`

Files		
Interface /primitives	**Description**	**Implémentation en objet (fonctions ou méthodes)**
creer_file() -> File	Créer une file vide	`ma_file = File()`
est_file_vide(`f`) -> Booléen	Teste si la file `f` est vide	`mon_booléen = ma_file.est_file_vide()`
enfiler(`f` , `élément`)	Insère `élément` en fin de `f`	`ma_file.enfiler(élément)`
defiler(`f`) → `élément`	Enlève l'`élément` en tête de la file `f` et le renvoie	`élément_tête = ma_pfile.défiler()`
tête(`f`) -> `élément`	Renvoie l'`élément` en tête de la file `f`	`élément_tête = ma_file.tête()`

Listes (chaînées)

A ne pas confondre avec le type List de Python, qui en fait caractérise les tableaux dynamiques.

Une liste est un enchaînement de cellules, chaque cellule étant composée d'une tête et d'une queue. La tête comporte la valeur de la cellule. La queue est soit une autre liste (donc pointant sur une cellule), soit vide si on se situe en fin de liste.

Listes

Interface /primitives	**Description**	**Implémentation en objet (fonctions ou méthodes)**
creer_liste() -> Cellule	Créer une liste vide	`ma_liste = Cellule()`
est_liste_vide(p) -> Booléen	Teste si la liste `liste` est vide	`mon_booléen = ma_liste.est_liste_vide()`
ajouter_élément(`liste,élément`)	Insère `élément` en tête de `liste`	`ma_liste = Cellule(élément, ma_liste)`

donnée(`liste`) → `élément` **ou** `None`
tête(`liste`) → `élément` **ou** `None`
	Renvoie l'`élément` en tête de la liste `liste` (deux formulations possibles tête ou donnée)	`# c'est un attribut :`
		`tête = ma_liste.tête`
		`donnée = ma_liste.donnée`

suivante(`liste`) -> `liste` **ou** `None`
queue(`liste`) -> `liste` **ou** `None`
	Renvoie a `liste` `suivante`, en queue de `liste` (deux formulations possibles queue ou suivante)	`# c'est un attribut :`
		`queue = ma_liste.queue`
		`suivante = ma_liste.suivante`

BASES DE DONNEES

Modèle relationnel :
Table = { *clé_primaire* : `type`, *#clé_étrangère* : `type`, *attribut_1* : `type`, etc.}
On dit table ou relation ou schéma relationnel

Contraintes :
- Entité : en lien avec la notion de clé primaire. Permet de s'assurer que chaque relation est unique.
- Référence : en lien avec la notion de clé étrangère, qui doit être créée avant le référencement dans un attribut d'un enregistrement.
- Domaine : en lien avec le type des attributs.
- Utilisateur : rajoutée par l'utilisateur, penser en particulier aux numéros de téléphone qui sont des chaînes et non des entiers, puisque commençant par + ou par 0.

Requêtes SQL :
- Exploration des données, version programme de Terminale NSI
  SELECT *liste d'attributs* (et *fonctions d'agrégation*)
  FROM *table*
  INNER JOIN *table* ON *égalité d'attributs*
  WHERE *condition(s)*
  ORDER BY *attributs de tri* (chaque attribut suivi de DESC ou ASC)
- Ajout d'un enregistrement
  INSERT INTO *table* (*liste d'attributs facultative*) VALUES *liste de valeurs*
- Modification d'un ou plusieurs enregistrements
  UPDATE *table* SET *att1 = val1*, *att2 = val2*,… (WHERE *condition*)
- Suppression d'un ou plusieurs enregistrements
  DELETE FROM *table* WHERE *condition*

Critères ACID pour les SGBD :
- Atomicité : une transaction se fait au complet ou pas du tout (efficacité de traitement des requêtes)
- Cohérence : cette propriété assure que chaque transaction amènera le système d'un état valide à un autre état valide (efficacité de traitement des requêtes).
- Isolation : les transactions s'exécutent comme si elles étaient seules sur le système (gestion des accès concurrents).
- Durabilité : une fois qu'une transaction a eu lieu, la base de données reste dans l'état modifié (persistance des données).

Un SGBD permet que l'accès aux BD soit sécurisé, avec des droits différents suivant les utilisateurs (sécurisation des accès).

Structures de Données Hiérarchiques

Vocabulaire des arbres binaires : racine, nœud, feuille, sous-arbre gauche, sous-arbre droit, taille, hauteur, arbre parfait, arbre complet
Attention à l'ambiguïté sur la notion de hauteur : un arbre vide peut être considéré de hauteur 0 ou -1 suivant les définitions/sujets.
De même il y a ambiguïté sur les notions d'arbres parfaits et d'arbres complets.

Lien entre taille n et hauteur h :
Ici la hauteur d'un arbre vide est -1

$$h+1 \leq n < 2^{h+1} \Leftrightarrow \log_2 n - 1 < h \leq n-1$$

Algorithmes : taille et hauteur

Calcul de la taille : retourne le nombre de sommets de l'arbre (racine + nœuds + feuilles)

Fonction récursive taille(*arbre*) :
  Si *arbre* est vide
    Retourner 0
  Sinon
    Retourner 1 + taille (*fils gauche*) + taille (*fils droit*)

Calcul de la hauteur : retourne la hauteur de l'arbre

Fonction récursive hauteur(*arbre*) :
  Si *arbre* est vide
    Retourner 0
  Sinon
    Retourner 1 + max(hauteur (*fils gauche*), hauteur(*fils droit*))

Algorithmes : parcours en profondeur

i. Parcours préfixe : racine – gauche – droit (la racine avant les enfants, la racine *pré*cèdent)
ii. Parcours infixe : gauche – racine – droit (la racine est au milieu, « *in* »)
iii. Parcours suffixe : gauche – droit – racine (la racine est en dernier, la racine *su*ccède)

Préfixe	Infixe	Suffixe

Fonction visitePréfixe(arbre) :
  Si arbre n'est pas vide :
  visiter racine
  visitePréfixe (fils gauche)
  visitePréfixe (fils droit)
	Fonction visiteInfixe(arbre) :
	  Si arbre n'est pas vide :
	  visiteInfixe (fils gauche)
	  visiter racine
	  visiteInfixe (fils droit)
		Fonction visiteSuffixe(arbre) :
		  Si arbre n'est pas vide :
		  visiteSuffixe (fils gauche)
		  visiteSuffixe (fils droit)
		  visiter racine

Algorithmes : parcours en largeur

On utilise une file :

1 . Mettre le nœud source dans la file.
2. Retirer le nœud du début de la file pour le traiter, c'est-à-dire afficher sa valeur.
3. Mettre le fils gauche et le fils droits lorsqu'ils sont non vides, non explorés, à la fin de la file.
4. Si la file n'est pas vide reprendre à l'étape 2.

Arbres binaires de recherche (ABR)
Définition : Pour tout nœud x, tous les nœuds situés dans le sous-arbre gauche de x ont une valeur inférieur ou égale à celle de x, et tous les nœuds situés dans le sous-arbre droit ont une valeur supérieure ou égale à celle de x.

Algorithmes sur les arbres binaires de recherche : recherche d'un élément

Fonction récursive recherche(*ABR, valeur*) :

```
    Si ABR est vide
        Retourner None                  # variante : retourner Faux
    Sinon
        valeur(x)  = étiquette(ABR)     # valeur de la racine
        Si valeur < valeur(x) :
            Retourner recherche(fils_gauche , valeur)
        Sinon si valeur > valeur(x) :
            Retourner recherche(fils_droit , valeur)
        Sinon
            Retourner ABR               # variante : retourner Vrai
```

Complexité en $O(\log n)$, si l'arbre est équilibré

Algorithmes sur les arbres binaires de recherche : insertion d'un élément

Fonction récursive ajoute(*ABR, valeur*) :

```
    Si ABR est vide
        renvoyer Arbre(valeur, None, None)
    Sinon si valeur <= étiquette_noeud(ABR)
        renvoyer  Arbre(étiquette_noeud(ABR),  ajoute(fils_gauche , valeur),  fils_droit)
    Sinon
        renvoyer Arbre(étiquette_noeud(ABR),  fils_gauche ,  ajoute(fils_droit , valeur))
```

Complexité en $O(\log n)$, si l'arbre est équilibré

Réseaux et protocoles de routage

Adresses IP et masques de sous-réseaux : sur une IP du type *xxx.yyy.zzz.ttt*, les nombres varient de 0 à 255. Le masque de sous-réseau 255.255.255.0 signifie que l'adresse du réseau est *xxx.yyy.zzz.*0, l'adresse de broadcast est *xxx.yyy.zzz.*255 et les machines ont une adresse comprise entre *xxx.yyy.zzz.*1 et *xxx.yyy.zzz.*254. Toute machine n'ayant pas la même adresse de réseau (les trois premiers nombres *xxx.yyy.zzz*) devra passer par un routeur pour communiquer avec une machine de ce réseau.
Remarque : n'est pas officiellement au programme mais présent dans de nombreux sujets.

Protocole RIP : la métrique est le nombre de sauts, ou bien le nombre de trajets entre routeurs. Les deux quantités diffèrent de 1. Efficace sur un petit réseau de 10 routeurs ou moins.

Protocole OSPF : la métrique est donnée par la vitesse du réseau, en général calculée par $\frac{10^8}{\text{bande passante du lien en bit/s}}$. Efficace jusqu'à 100 routeurs.

Diviser pour Regner

Principe :

- Diviser : partager le problème en sous-problèmes de même nature (souvent de taille n /2)
- Régner : résoudre ces différents sous-problèmes (presque toujours en récursif)
- Combiner : fusionner les solutions pour obtenir la solution du problème initial

Comprendre l'algorithme du tri fusion

Programmation dynamique

Principe :

- Trouver une formule de récurrence pour obtenir la valeur optimale
- Écrire un algorithme itératif (et non pas récursif) pour obtenir l'optimum. On part des plus petits problèmes et on va vers les plus grands. Cette partie donne la valeur optimale, mais pas la manière dont elle est obtenue.
- Reconstruire la solution optimale a posteriori. Cette dernière étape est parfois ignorée dans le cas de problèmes avec des données très lourdes. On part du plus grand problème et on redescend vers les petits.

Comprendre l'algorithme du rendu de monnaie

Différences entre diviser pour régner et programmation dynamique :

- Diviser pour régner : on part du problème complet, et on le divise en petits morceaux. L'approche est descendante, du haut vers le bas (en anglais : top-down)
- Programmation dynamique : on résout des petits problèmes que l'on combine pour arriver au problème complet. L'approche est ascendante, du bas vers le haut (en anglais bottom-up)
- Diviser pour régner : les « petits problèmes » sont indépendants.
- Programmation dynamique : les « petits problèmes » sont dépendants.
- Diviser pour régner : algorithmes récursifs
- Programmation dynamique : algorithmes itératifs.

Structures de Données Relationnelles : Les Graphes

Vocabulaire : sommet (étiquetés ou non) , arcs/arêtes (pondérés/valués ou non), sommets voisins/ascendants/descendants/successeurs/prédécesseurs, degré, chemin/chaîne (élémentaire/simple ou non), cycle/circuit

Implémentation : matrice d'adjacence, dictionnaire de listes de successeurs, objet

Algorithmes : parcours en profondeur (*algorithme non déterministe)*

Algorithme : *descente en profondeur*
Donnée : un graphe *G* et un sommet *s* de *G*
Résultat : les sommets marqués sont les descendants de *s* dans *G*
Ici les sommets sont marqués Faux/Vrai suivant s'ils ont déjà été visités ou non, on crée un tableau de booléens pour sauver ces marques.

```
P ← NouvellePile()
Empiler(P, s)
Tant que Non EstPileVide(P) faire
    u ← Dépiler(P)        # on enlève la tête de la pile
    Si u n'est pas marqué
        Marquer u
        Afficher u        # ou le sauver dans une structure
        Pour chaque v voisin de u faire
                Empiler(P, v)
```

Algorithme : parcours en largeur (*algorithme non déterministe)*

Algorithme : *descente_en_largeur*
Donnée : un graphe *G* et un sommet *s* de *G*
Résultat : les sommets marqués sont les descendants de *s* dans *G*
Ici les sommets sont marqués Faux/Vrai suivant s'ils ont déjà été visités ou non, on crée un tableau de booléens pour sauver ces marques.

```
F ← NouvelleFile()
Enfiler(F, s)
Marquer s
Tant que Non EstFileVide(F) faire
    u ← Défiler(F)        # on enlève le 1er élément de la file
    Afficher u            # ou le sauver dans une structure
    Pour chaque v voisin de u faire
        Si v n'est pas marqué alors
            Marquer v
            Enfiler(F, v)
```

Algorithmes : recherche d'un cycle (*algorithme non déterministe)*

Algorithme au programme, assez difficile à comprendre. Construit à partir du parcours en profondeur, à comprendre en le faisant tourner à la main sur des exemples.

Algorithme : *descente en profondeur, recherche de cycle dans un graphe non orienté.*
Données : un graphe *G* connexe. Sommet de départ
Résultat : existence ou non d'un cycle dans *G*

```
Prédécesseur ← dictionnaire dont les clés sont les sommets et les valeurs -1
P ← NouvellePile()
Empiler(P, départ)
Prédécesseur[départ] ← départ
Tant que Non EstPileVide(P) faire
    u ← Dépiler(P)
    Pour chaque v voisin de u faire
        Si Prédécesseur[v] = - 1 alors
            Prédécesseur[v] ← u
            Empiler(P, v)
        Sinon si Prédécesseur[u] ≠ v alors
            Renvoyer Vrai
Renvoyer Faux
```

Algorithmes : recherche d'un chemin le plus court entre deux sommets (*algorithme non déterministe)*

Cet algorithme n'est pas clairement au programme explicitement. Au programme il y a « recherche de chemin dans un graphe » ; autant rechercher un chemin le plus court possible. Comprenez-le : c'est une adaptation du parcours en largeur.

Algorithme : *descente en largeur avec distances, graphe non orienté*
Donnée : un graphe *G* et un sommet *s* de *G*
Résultat : La fonction *d* donne la distance de *s* à chaque sommet de *G*
Dans cet algorithme, les sommets visités ne sont pas marqués, mais ils ont une distance finie au sommet d'origine (cf. remarque sur la similitude avec BFS)

```
# on crée une structure de données d pour stocker les distances
Pour u dans l'ensemble des sommets de G faire d(u) = +∞
d(s) = 0
F ←  NouvelleFile()
Enfiler(F, s)
Tant que Non EstFileVide(F) faire
    u ← Défiler(F)
    pour chaque v voisin de u faire
        si d(v) = +∞ alors
            Enfiler(F, v)
            d(v) = d(u) + 1
```

Sécurité des communications

Vocabulaire : chiffrement symétrique (exemple de nom de protocole à connaître AES), chiffrement asymétrique (exemple de nom de protocole à connaître RSA), certificat de sécurité

Fonctionnement du protocole HTTPS : génération de la clé symétrique via un protocole asymétrique et certification via certificat de sécurité dans un premier temps, échange des données via protocole symétrique pour l'échange des données dans un second temps.

LES PROCESSUS

Commandes linux de base :
`ls -a -l, mkdir, cd, touch, rm, rmdir, ps -a -l, kill`

Identifiants : PID (identifiant du processus), PPID (identifiant du processus parent)

Etats des processus :

- prêt
- chosen/élu/en éxecution)
- sleeping/blocked/en attente
- terminé

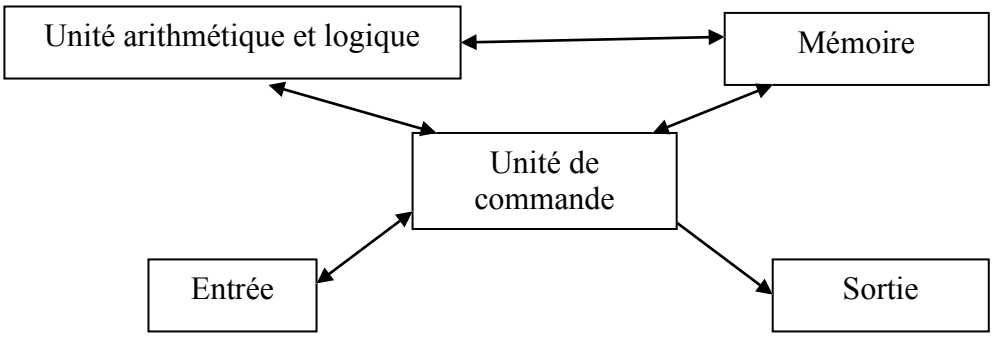
Interblocage : comprendre comment un interblocage peut advenir (conditions de Coffman facultatives)

Calculabilité – Décidabilité

Un programme est une donnée.
Tous les langages de programmation permettent de *calculer* les mêmes résultats (thèse de Church Turing).
Certains problèmes informatiques ne sont pas calculables, ils sont *indécidables*. Notamment le problème de l'arrêt : il n'existe pas de programme permettant de déterminer si un programme passé en paramètre s'arrête sur des données précises, passés également en paramètres.

SYSTÈMES SUR PUCE

Retenir les deux schémas du cours de 1ère :

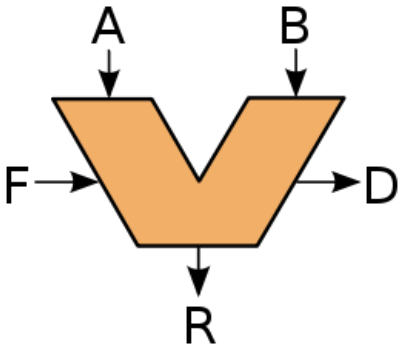


UAL

L'unité arithmétique et logique est le cœur de l'ordinateur.

On la représente habituellement par le schéma ci-contre :

- A et B sont les opérandes (les entrées)
- R est le résultat
- F est une fonction binaire (l'opération à effectuer)
- D est un drapeau indiquant un résultat secondaire de la fonction (signe, erreur, division par zéro, « supérieur »,…)



Avantages des SOC :

- Amélioration des performances.
- Limitation de l'augmentation de température.
- Coûts globalement limités
- Adaptation aux besoins spécifiques.

Inconvénients des SOC :

- Maintenance matérielle impossible (au mieux très complexe).
- Technicité de la conception et de la fabrication.
- Machine figée et non améliorable.

Recherche textuelle Boyer Moore

Principe : recherche d'un *motif* dans un *texte* grâce à une *fenêtre* flottante.

Comprendre :
- la règle du mauvais caractère.
- la construction du dictionnaire des positions des caractères du motif.
- les règles de décalage de la fenêtre suivant la position du mauvais caractère.

Savoir faire tourner l'algorithme de Horspool à la main.
Remarque : l'ensemble de la communauté éducative est d'accord sur le fait que l'algorithme de Boyer-Moore proprement dit est trop complexe pour le niveau de terminale (même s'il est officiellement au programme).