

1 Programme du premier semestre

Les séances de travaux pratiques du premier semestre poursuivent les objectifs suivants :

- consolider l'apprentissage de la programmation en langage Python qui a été entrepris dans les classes du lycée;
- mettre en place un environnement de travail;
- mettre en place une discipline de programmation : spécification précise des fonctions et programmes, annotations et commentaires, jeux de tests;
- introduire les premiers éléments de complexité des algorithmes : on ne présente que l'estimation asymptotique du coût dans le cas le pire;
- introduire des outils de validation : variants et invariants.

Le tableau ci-dessous présente les thèmes qui sont abordés lors de ces séances, et, en colonne de droite, une liste, sans aucun caractère impératif, d'exemples d'activités qui peuvent être proposées aux étudiants. L'ordre de ces thèmes n'est pas impératif.

Aucune connaissance relative aux modules éventuellement rencontrés lors de ces séances n'est exigible des étudiants.

Thèmes	Exemples d'activité, au choix du professeur et non exigibles des étudiants. Commentaires.
Recherche séquentielle dans un tableau unidimensionnel. Dictionnaire.	Recherche d'un élément. Recherche du maximum, du second maximum. Comptage des éléments d'un tableau à l'aide d'un dictionnaire. <i>Manipulations élémentaires d'un tableau unidimensionnel. Utilisation de dictionnaires en boîte noire. Notions de coût constant, de coût linéaire.</i>
Algorithmes opérant sur une structure séquentielle par boucles imbriquées.	Recherche d'un facteur dans un texte. Recherche des deux valeurs les plus proches dans un tableau. Tri à bulles. <i>Notion de complexité quadratique. On propose des outils pour valider la correction de l'algorithme.</i>
Utilisation de modules, de bibliothèques.	Lecture d'un fichier de données simples. Calculs statistiques sur ces données. Représentation graphique (histogrammes, etc.).
Algorithmes dichotomiques.	Recherche dichotomique dans un tableau trié. Exponentiation rapide. <i>On met en évidence une accélération entre complexité linéaire d'un algorithme naïf et complexité logarithmique d'un algorithme dichotomique. On met en œuvre des jeux de tests, des outils de validation.</i>
Fonctions récursives.	Version récursive d'algorithmes dichotomiques. Fonctions produisant à l'aide de <code>print</code> successifs des figures alphanumériques. Dessins de fractales. Énumération des sous-listes ou des permutations d'une liste. <i>On évite de se cantonner à des fonctions mathématiques (factorielle, suites récurrentes). On peut montrer le phénomène de dépassement de la taille de la pile.</i>
Algorithmes gloutons.	Rendu de monnaie. Allocation de salles pour des cours. Sélection d'activité. <i>On peut montrer par des exemples qu'un algorithme glouton ne fournit pas toujours une solution exacte ou optimale.</i>
Matrices de pixels et images.	Algorithmes de rotation, de réduction ou d'agrandissement. Modification d'une image par convolution : flou, détection de contour, etc. <i>Les images servent de support à la présentation de manipulations de tableaux à deux dimensions.</i>
Tris.	Algorithmes quadratiques : tri par insertion, par sélection. Tri par partition-fusion. Tri rapide. Tri par comptage. <i>On fait observer différentes caractéristiques (par exemple, stable ou non, en place ou non, comparatif ou non, etc).</i>

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 <code>collections.deque</code> . 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.