TP2 : Premiers algorithmes en Python

CPES ≪ Sciences des données, arts et cultures ≫ : Introduction à l'algorithmique

Antoine GAUQUIER
antoine.gauquier@ens.fr

Pierre SENELLART
pierre@senellart.com

19 septembre 2023

Dans ce deuxième TP, vous allez appliquer les notions abordées lors du précédent cours, et implémenter vos premiers algorithmes en langage Python. Il vous sera également demandé dans certains exercices de d'abord écrire l'algorithme en **pseudo-code**, c'est à dire, indépendemment de tout langage de programmation. Il vous faudra ensuite implémenter ces algorithmes en langage Python. Il est attendu que vous utilisiez les notions vues lors du précédent cours. En particulier, les algorithmes doivent tous être définis au sein de fonctions, et les tests de ces fonctions doivent faire partie d'une fonction `main`, tel que décrit dans le cours.

Avant d'entamer ce nouveau TP, nous commencons par corriger l'exercice 4 du précédent TP, qui n'avait pas pu être traité.

Exercice 1: Méthode de Monte Carlo.

La *méthode de Monte Carlo* est une méthode numérique permettant d'obtenir une valeur approchée de certains quantités. Comme les casinos de Monte Carlo, elle repose sur le tirage d'un grand nombre de valeurs aléatoires.

On peut notamment l'utiliser pour estimer la valeur de π. En effet, on sait que la surface du disque unité, défini par l'ensemble des points $(x, y) \in [-1, 1]^2$ tels que $x^2 + y^2 \leqslant 1$, vaut π. Il suffit donc de tirer aléatoirement un nombre suffisamment grand de couples (x, y) dans $[0, 1]^2$ afin d'avoir une valeur approchée de $\frac{\pi}{4}$, et en la multipliant par 4, de π.

Question 1. Écrire une fonction Python `monte_carlo_pi` prenant en argument un entier positif n décrivant le nombre de couples à tirer aléatoirement, et retournant une valeur approchée de π à partir de ces tirages, en utilisant la méthode de Monte Carlo. *Note : On pourra utiliser la fonction* `random` *du package* `random` (avec `import random`) *afin de générer des nombres aléatoires entre 0 et 1.*

Question 2. Tester votre fonction avec plusieurs puissances de 10 pour n. Cette méthode est-elle précise ?

Question 3. En utilisant une boucle, déterminer un ordre de grandeur de n à partir duquel la valeur estimée de π est distante de moins de 10^{-3} de la vraie valeur de π (dont une approximation comme nombre en virgule flottante est donnée par `math.pi`)

Exercice 2: Recherche dans une liste ordonnée.

On considère une liste l de nombres entiers, rangés par ordre croissant. On souhaite maintenant savoir si le nombre x est contenu dans la liste l, et si oui, sa position dans cette liste. Si x est présent plusieurs fois, alors on retournera la position du premier x rencontré.
Une méthode « naïve » permettant d'y arriver est de parcourir l'intégralité de la liste, en comparant chaque élément de l avec x. Si on a égalité, alors x est dans la liste, et on retourne sa position. Si il n'y a pas eu d'égalité une fois toute la liste parcourue, alors x n'est pas dans l (et on peut retourner une valeur par défaut décrivant l'absence de x : -1 par exemple). Cette méthode est présentée dans la slide 10 du cours du 12 septembre.
Bien que fonctionnelle, elle est assez coûteuse : elle requiert de parcourir toute la liste l, et donc de faire (au pire cas) $|l|$ comparaisons. On parlera alors de *complexité temporelle* linéaire, qu'on pourra noter $O(|l|)$ (cette notion sera présentée plus tard dans le cours). Cela signifie que si l contient 1 000 000 d'entrées, alors il faudra effectuer 1 000 000 de comparaisons. On souhaite donc trouver une méthode plus efficace, à savoir, une méthode dont la *complexité temporelle* est plus faible.
On propose pour cela d'utiliser la technique dite du « *diviser pour régner* ». *Diviser pour régner* est une technique algorithmique qui se déroule en trois phases. D'abord *diviser*, qui consiste à découper un problème initial en plusieurs sous-problèmes, plus « simples » (au sens large du terme) à résoudre. Ensuite, *régner* consiste à résoudre chacun de ces sous-problèmes. Finalement, *combiner*, qui consiste, à partir de la solution de chaque sous-problème, à résoudre le problème initial.
L'application de la technique « *diviser pour régner* » au problème de la recherche dans la liste ordonnée nous donne l'algorithme de la *recherche dichotomique*. Voici les étapes de cet algorithme. On commence par déterminer la valeur de l'élément central de la liste (ou presque central, si la taille de la liste est paire), qui correspond à `valeur_centrale` $= l\left[\left\lfloor \frac{|l|}{2} \right\rfloor\right]$. Si `valeur_centrale` $> x$, alors, comme la liste est triée, on sait que si x est contenu dans l, il se trouvera nécessairement dans la moitié inférieure de la liste. Sinon, si `valeur_centrale` $< x$, alors il se trouvera dans la moitié supérieure de la liste. Bien entendu, si on a `valeur_centrale` $= x$, alors on l'a trouvé, et on peut retourner sa position.

Question 1. En utilisant la description de l'algorithme de la *recherche dichotomique*, écrire le **pseudo-code** d'un algorithme **récursif** qui effectue cette recherche.
Indice : Lorsque vous faites un appel récursif de la fonction, et que vous lui passez en paramètre une sous-liste, la sous-liste en question est (de fait) ré-indexée... Il vous faut donc vous-même conserver l'indice du premier élément de la sous-liste dans la liste d'origine d'une façon ou d'une autre. Exemple : si j'ai la liste $[1, 3, 4, 6, 7]$*, et que je considère la sous-liste* $[6, 7]$ *dans un appel récursif, alors la position de* 6 *sera* 0 *dans cet appel, alors qu'en réalité, dans la liste de la fonction à l'origine de l'appel, sa position est* 3.

Question 2. Implémentez ensuite cet algorithme en langage Python. Écrivez une portion de code permettant de tester votre algorithme, en envisageant différents cas de figure ($x \notin l$, $|l| = 0$, etc.).

Exercice 3: Algorithme d'Euclide.

L'algorithme d'Euclide est un algorithme permettant le calcul du *plus grand commun diviseur* (PGCD) de deux entiers passés en entrée. Concrètement, le PGCD est le plus grand entier qui divise les deux entiers passés en entrés (donc, le plus grand entier pour lequel le reste des divisions euclidiennes avec les deux entiers passés en entrée sont nuls).

Question 1. A partir de l'*organigramme de programmation* (parfois aussi appelé « algorigramme ») présenté en Figure 1, implémenter l'algorithme d'Euclide **en pseudo-code**.

Question 2. Une fois le pseudo-code déterminé, écrire l'algorithme d'Euclide en Python. Vérifier avec les paires d'entiers suivantes : $(126, 297)$, $(1231, 754)$ et $(26187, 11223)$.

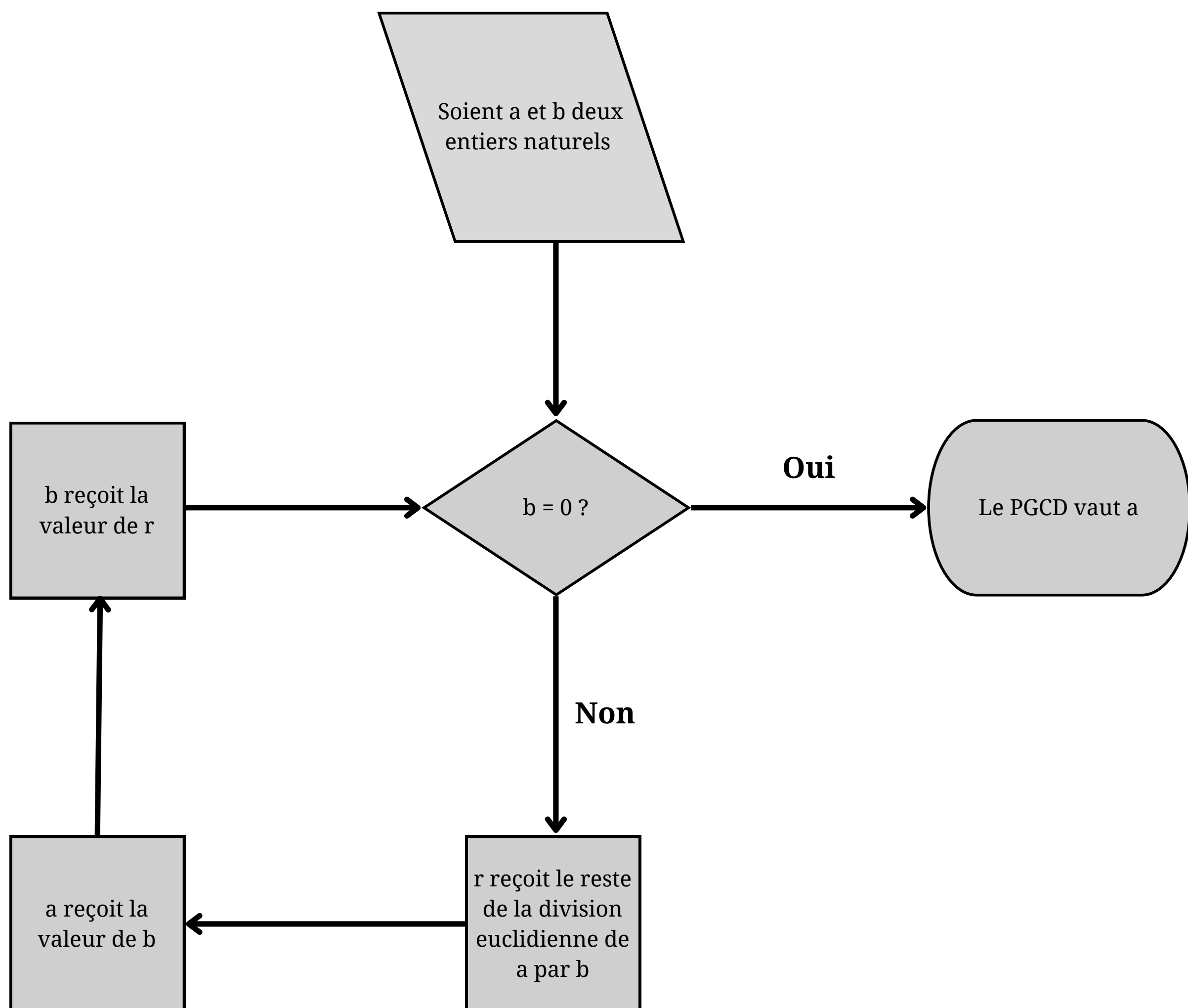

FIGURE 1 – Algorigramme de la version itérative de l'algorithme d'Euclide.