Ce que peut apprendre une pratique de la classe

Bernard Egger(*)

J'enseigne en classe préparatoire ECE. Pour ceux qui ignorent ce que signifie ce sigle, il s'agit de « économique et commerciale, voie E (pour ES) ». C'est-à-dire ce que l'on appelle plus communément les classes préparatoires HEC.

L'épreuve que passeront mes étudiants aux concours s'appelle « Maths et informatique ». Elle contient en effet une partie informatique (sur papier nécessairement puisque tout instrument de calcul est interdit, même une calculatrice « 4 opérations »). Il s'agit toujours d'un programme à écrire, à compléter ou à interpréter. Le langage de programmation utilisé est le Turbo-Pascal.

Comme chacun sait, c'est un langage plutôt ancien, mais bien adapté à une utilisation pédagogique. Du moins tant que l'on n'essaie pas de tester les programmes...

Survivance du DOS et des systèmes d'exploitation de même type, la saisie des programmes comme leur exécution se font dans un environnement vieillot, peu en rapport avec celui que connaissent au quotidien nos élèves.

Comme toujours en programmation, la moindre erreur de syntaxe est fatale comme par exemple l'oubli (fréquent) d'un point-virgule... Sur une feuille de papier, c'est évidemment moins gênant.

Les concours n'évaluent les connaissances informatiques que par écrit, mais il n'en reste pas moins que l'on aboutit à une situation commune quand on enseigne un langage de programmation à des étudiants pour lesquels il n'occupera qu'une place très modeste dans leur formation : on perd un temps inutile avec la syntaxe souvent au détriment d'une compréhension plus fine de l'algorithme.

Pour que le programme « tourne », il faut faire attention à tout, et cela prend beaucoup de temps surtout en début d'apprentissage. Après quelques séances, on se lasse et l'on se contente de « faire tourner » le programme au tableau (on se rassure alors en se disant que c'est pédagogiquement efficace puisque les étudiants seront obligés de procéder de cette façon dans les épreuves de concours).

La programmation est sans aucun doute une école de rigueur, mais une trop grande « rigidité » nuit à la compréhension des enjeux réels de l'intérêt de la programmation, c'est-à-dire l'élaboration d'algorithmes.

Que cherche-t-on en définitive dans ce type de classes ? Évidemment pas à fabriquer des « programmeurs », mais plutôt à faire comprendre aux étudiants que l'informatique peut compléter efficacement la démarche mathématique en fournissant des résultats qui souvent l'éclairent.

Prenons l'exemple des suites récurrentes. Elles constituent un des aspects essentiels des exercices de programmation demandés dans les concours. L'étude

(*) egger.bernard@wanadoo.fr

mathématique d'une telle suite vise le plus souvent à justifier sa convergence (ou sa divergence). Nous connaissons l'importance de ce résultat dans la résolution d'équations pour lesquelles l'on ne sait pas donner la (ou les solutions) sous forme exacte. L'enjeu est d'autant plus important puisque c'est de cette façon que fonctionnent les solveurs des calculatrices ou des tableurs.

Un autre type de programmation largement proposé aux concours est celui de la simulation de situations aléatoires : programmation d'expériences aléatoires, de variables aléatoires, … , illustration de théorèmes. Se convaincre de la loi faible des grands nombres ou du théorème central limite, alors qu'à ce niveau aucune démonstration n'est pas vraiment possible, est très largement facilité par des simulations de « grande ampleur » comprenant un grand nombre de données.

Mais « plus simplement », il est sans doute tout aussi important de faire comprendre que l'obtention de 15 piles consécutifs quand on jette une pièce bien équilibrée n'est pas un événement en opposition avec la loi des grands nombres. Réaliser manuellement l'expérience risque d'être laborieux (si l'on manque de chance, il ne sera pas rare d'avoir à réaliser plus de 30 000 lancers). Mais est-on obligé de programmer une telle situation ? Avec une calculatrice, sans aucun doute. Un tableur permettra sans doute dans un premier temps d'éviter la programmation, mais, dans tous les cas, la simulation de la situation n'est-elle pas déjà le résultat d'un algorithme ?

Le tableur ne permettra d'ailleurs pas d'éviter complètement la programmation. Si 30 000 lancers peuvent suffire pour obtenir 15 piles consécutifs (et bien souvent beaucoup plus), qu'en est-il du cas de 20 piles consécutifs ? On dépasse souvent le million de lancers. Nous y reviendrons plus loin.

Avec un peu d'habitude, l'élaboration d'un algorithme de suites récurrentes, ou celui correspondant à la situation précédente, est rapidement à la portée de mes étudiants.

Sa traduction en Turbo-Pascal reste l'étape délicate.

L'apprentissage de l'algorithmique peut paraître aux antipodes de celui de l'algèbre.

En algèbre, l'entraînement préalable sur des techniques qui ne font pas sens est une condition de la libération future de l'imagination.

En programmation, l'imagination est là d'abord ; l'écriture du programme apparaît comme la partie rébarbative, « insensée » de la démarche. Le problème est qu'elle est postérieure à ce qui est intéressant, enrichissant

Algorithmique et programmation

Des algorithmes, il y en a à tous les coins de toutes les rues mathématiques (mais pas seulement dans ce monde-là). Construire la médiatrice d'un segment à la règle et au compas relève d'un algorithme très clair (au même titre d'ailleurs que toute démarche constituée par la succession bien répertoriée de « gestes », de la recette de cuisine à l'analyse grammaticale d'une phrase). On pourrait envisager de nommer algorithme de la médiatrice la « liste des instructions » qu'il faut suivre pour construire une médiatrice. Et dans le même ordre d'idée, ne pourrait-on pas nommer

« algorithme du triangle rectangle » la suite d'opérations (qui s'appuient sur la réciproque du théorème de Pythagore) permettant de dire si un triangle est rectangle ou pas quand on connaît ses côtés ?

L'usage mathématique a voulu réserver le nom d'algorithme à un autre type de situation comme par exemple l'algorithme d'Euclide (ou à celui des soustractions), ou encore à l'algorithme de Dijkstra en théorie des graphes (le « premier » d'une longue série d'algorithmes dans cette théorie).

Prenons l'exemple de l'algorithme d'Euclide universellement connu.

Son énoncé n'est pas plus difficile que la plupart des démarches algorithmiques de construction de figures en géométrie. On peut même dire qu'il est particulièrement simple. Il est utile de le rappeler ici : pour déterminer le PGCD de deux nombres entiers a et b, on effectue la division euclidienne de a par b. Si le reste r de cette division est nul, alors PGCD$(a,b) = b$; sinon on recommence avec b et r. Ce qui apparaît clairement ici, comme dans la plupart des algorithmes « mathématiques », c'est la notion de boucle.

L'algorithme « mathématique » n'est pas exactement une suite finie d'instructions permettant selon des conditions précises d'aboutir à un but fixé à l'avance. C'est davantage une **économie de pensée**, comme l'est la notion de boucle et l'une de ses traductions mathématiques : la récurrence.

La liste des instructions nécessaires à la construction d'une figure géométrique est de nature algorithmique, mais peut-elle être assimilée à un algorithme au sens « mathématique » ?

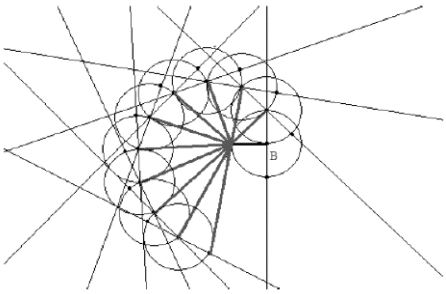
Dans ce cas, un nombre important de situations d'apprentissage intra ou extra mathématiques auront le même statut.

Il est sans doute utile de décrire ce type de situations sous forme d'algorithme, mais il n'est pas sûr qu'un usage trop abondant de ce type de description n'apporte pas une grande confusion.

Mettre en évidence l'économie que suppose l'utilisation mathématique de la notion d'algorithme paraît nettement plus pertinent.

Ceci suppose de trouver des « situations de boucles et de conditions ».

Leur traduction sous forme d'un programme exprimé dans un langage informatique n'est pas nécessaire dans un premier temps. Une situation typique est celle de la construction à la règle et au compas des racines carrées des nombres entiers successifs à partir de 1. Il s'agit là d'une simple récurrence. Si l'on a un segment de longueur $\sqrt{n}$, on construit le segment de longueur $\sqrt{n+1}$ en traçant une perpendiculaire à ce segment en l'une de ses extrémités, puis en traçant un cercle de centre cette extrémité et de rayon 1. Le segment qui relie l'un des points d'intersection de ce cercle avec la perpendiculaire tracée précédemment avec l'autre extrémité du segment initial a pour longueur $\sqrt{n+1}$.



Remarquons que la répétabilité des diverses étapes de la construction conduit naturellement à la notion de macro. Cette notion est essentielle, car dans un logiciel de géométrie dynamique, elle permet de mettre en évidence le contenu de chaque boucle.

Il paraît donc pertinent de réserver le mot « algorithme » à des situations pour lesquelles cette dénomination indique un contexte particulier que j'ai désigné un peu génériquement sous le terme « économie de pensée ». Les boucles en sont des représentants importants. Mais cette limitation nous ramène évidemment à la programmation. Bien sûr, on peut faire « tourner » l'algorithme d'Euclide à la main, mais on se limitera alors à de petits nombres. Son efficacité, sa puissance ne seront plus vraiment visibles. Son utilité est moins évidente.

De la même façon, la construction géométrique des racines carrées des nombres entiers successifs prend sa pleine signification avec la notion de macro.

Les calcul des termes d'une suite récurrente à l'aide d'un tableur peut sembler échapper à ce constat. C'est sans doute parce que le tableur est construit autour de la notion de répétabilité qui est pratiquement identique à celle de boucle. C'est dans sa construction même que ce type de logiciel contient des boucles.

Il me semble donc très difficile de parler d'algorithme sans programmation, sous peine de réduire fortement l'intérêt d'une approche algorithmique.

Domaines de l'algorithmique dans les mathématiques du lycée

L'arithmétique assez naturellement, dans une moindre mesure la géométrie, fournissent des situations de mise en œuvre de la démarche algorithmique.

Il existe un autre champ privilégié : celui de la simulation en statistiques ou en probabilité.

L'exemple que nous avons donné plus haut sur les séries de 15 piles consécutifs met bien en évidence la nécessité de programmation d'une boucle pour répondre à la question.

Une situation très fréquente en probabilité est la répétition d'expériences de Bernoulli indépendantes et de même paramètre. Dans cette répétition, on s'intéresse habituellement soit au nombre de succès (loi binomiale), soit au nombre d'expériences qu'il faut réaliser pour aboutir à un succès. Les boucles « For » et « While » traduisent parfaitement ces situations sur le plan informatique. Dans une certaine mesure, elles en sont la traduction parfaite.

L'efficacité de la programmation ne réside pas simplement dans le fait d'obtenir un résultat en utilisant une boucle, mais aussi dans la possibilité de relancer le programme pour obtenir d'autres résultats.

Dans le cas d'une expérience aléatoire, ce qui importe est la reproductibilité de l'expérience. Obtenir facilement un grand nombre de résultats par de simples nouvelles exécutions d'un programme met en évidence l'intérêt de la programmation de l'expérience. C'est dans la proximité des résultats obtenus que l'on pourra sentir un ordre dans le hasard.

Le choix d'un langage

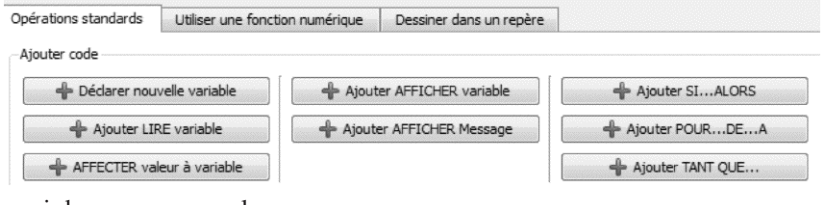
Mon expérience relatée au début de cet article m'a conduit à donner la préférence à la simplicité, même au détriment d'une certaine richesse.

En géométrie, pas de problème : tout logiciel de géométrie dynamique permettant de créer des macros convient.

Pour la programmation « véritable », ma préférence va à Algobox.

Construit sur le modèle du Turbo-Pascal, il a un premier avantage : être en français. Si l'on veut aller plus loin en programmation, l'utilisation de l'anglais sera indispensable, mais dans le cadre d'une initiation (qui sera celui de nos élèves de lycée), le français permet d'exprimer en « langage naturel » les différents moments de l'algorithme.

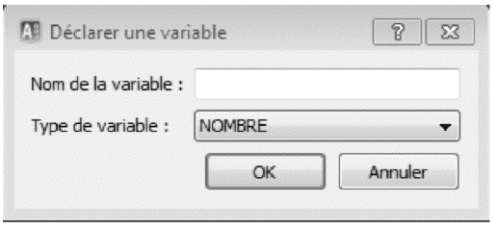
Un deuxième avantage, plus essentiel à mes yeux, est que la connaissance de la syntaxe est réduite au minimum. La plupart des instructions : boucles, conditions, affectations, affichage, … sont prises en charge directement par le logiciel.



Ce qui donne par exemple :

Enfin, Algobox oblige l'utilisateur à déclarer les variables présentes dans le programme ainsi que leur type (réduit à trois possibilités : nombre, chaîne, liste).

Cette contrainte qui n'apparaît pas sur les calculatrices ou en Visual Basic est très formatrice. On se rend compte bien vite qu'elle donne une plus grande intelligibilité à la lecture d'un programme fait par quelqu'un d'autre.



Si nous reprenons le programme sur les 15 piles successifs, il s'écrira avec Algobox :

```
1  VARIABLES
2      n EST_DU_TYPE NOMBRE
3      s EST_DU_TYPE NOMBRE
4      hasard EST_DU_TYPE NOMBRE
5      k EST_DU_TYPE NOMBRE
6  DEBUT_ALGORITHME
7      LIRE k
8      n PREND_LA_VALEUR 0
9      s PREND_LA_VALEUR 0
10     TANT_QUE(s!=k) FAIRE
11         DEBUT_TANT_QUE
12         n PREND_LA_VALEUR n+1
13         hasard PREND_LA_VALEUR random()
14         SI (hasard<0.5) ALORS
15             DEBUT_SI
16             s PREND_LA_VALEUR s+1
17             FIN_SI
18         SINON
19             DEBUT_SINON
20             s PREND_LA_VALEUR 0
21             FIN_SINON
22     FIN_TANT_QUE
23     AFFICHER n
24 FIN_ALGORITHME
```

Il n'est évidemment pas question d'aborder un tel programme en début d'apprentissage, mais il peut constituer un but pour des élèves qui ont un peu de recul. Il est d'ailleurs un peu plus général que le problème proposé puisqu'il permet de choisir le nombre de piles consécutifs que l'on veut obtenir.

Il existe de nombreux autres langages de programmation, mais en dehors d'Algobox, mon choix irait vers le Visual Basic d'Excel.

Tout d'abord, c'est le langage de programmation d'un tableur. À ce titre, il s'intègre naturellement dans l'environnement tableur, tout en le complétant quand ce dernier est inefficace, ce qui est le cas de notre problème (en pratique, malgré les

grandes possibilités qu'ils offrent, les tableurs sont relativement inefficaces quand il s'agit de répéter une instruction jusqu'à l'obtention d'une condition si cette condition apparaît « normalement » au bout d'un très grand nombre d'essais).

Visual Basic est également un langage riche en instructions (c'est un langage objet), dont la structure est très proche de celle du Pascal.

Enfin, il possède de nombreux outils pour « suivre un programme » pas à pas. Signalons en particulier la fonction Débogage.

Voici le même programme en VBA :

```
Function f(k)
Dim n, s, k as Integer
Dim hasard as Double
n=0
s=0
Do
        n = n + 1
        hasard = Rnd()
        If hasard < 0.5 Then
            s = s+1
        Else
            s=0
        End If
Loop While s <> k
f=n
End Function
```

Le programme est plus court qu'avec Algobox.

En l'écrivant sous forme de fonction, nous évitons la lecture de k, ce qui économise la déclaration de lecture de cette variable.

La boucle et les conditions s'expriment « plus simplement » (c'est-à-dire de façon moins détaillée) en Visual Basic.

La structure globale des deux programmes est identique et l'on passe facilement d'un langage à l'autre.

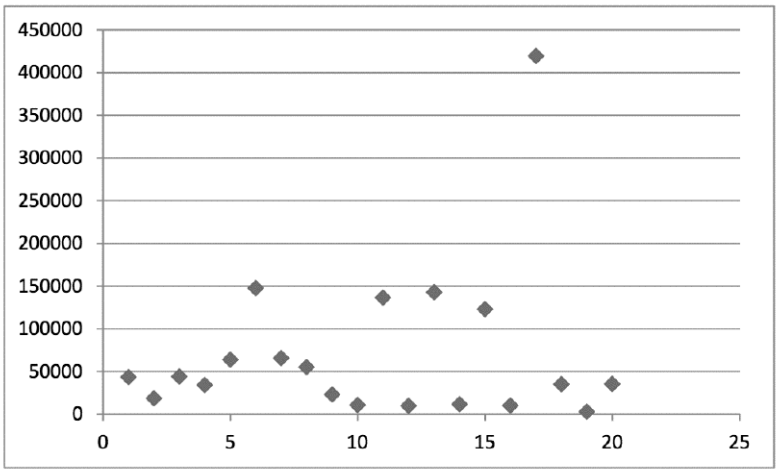
On obtient par exemple :

15	10 666
15	147 570
15	65 758
15	55 041
15	22 982

20	1 503 551
20	345 251
20	2 050 540
20	3 896 331
20	1 788 329

Les résultats sont très divers. Pour une série de 15 piles consécutifs, on trouve, sur cinq tentatives, entre 10 666 et 147 570 lancers. Pour 20 piles consécutifs, il en faut nettement plus.

L'intérêt du tableur est de disposer également d'un grapheur permettant ainsi d'obtenir directement un nuage de points plus explicite qu'un tableau de nombres.



Résultats pour 20 répétitions de lancers d'une pièce équilibrée jusqu'à obtention de 15 piles consécutifs.

Conclusion

Quelles que soient les difficultés que ne manquera pas de soulever l'introduction de l'algorithmique, je suis persuadé qu'il s'agit d'une bonne chose. Comme on l'aura remarqué dans cet exposé, je ne sépare pas cette introduction de l'usage de l'ordinateur. Même s'il existe un algorithme « papier-crayon » de résolution des équations du second degré (équations dont chacun sait qu'elles sont au début même de la notion d'algorithme), et même s'il est important de signaler le côté systématique de la démarche, il y a un risque d'être très réducteur si l'on appelle « algorithme » toute méthode pouvant se décrire comme une suite d'instructions. Cette notion dépasse ce cadre et ne peut vraiment se penser sans parler de boucles ou de conditions, concepts qui permettent une description simple de procédés complexes.

C'est l'ordinateur (ou la calculatrice dans un certain nombre de cas) qui donne la pleine mesure de l'efficacité de l'algorithme, et plus précisément la programmation. Les possibilités d'exécutions multiples et de tests sur de grandes valeurs permettent en particulier de mettre en évidence des régularités invisibles autrement. La notion d'invariant est centrale en mathématique, comme dans les autres domaines du savoir. Ce qui ne varie pas demande explication et souvent contribue au développement d'une théorie. Ce que l'informatique peut apporter, c'est de faciliter la recherche de certains invariants, aussi bien dans les logiciels de géométrie dynamique qu'avec l'utilisation de programmes. L'algorithme est alors la stratégie par laquelle on va d'abord se poser le problème, le programme sa mise en œuvre concrète. Par expérience, ce « second temps » doit être, au moins en début d'apprentissage, le plus simple possible. Il faut travailler sur des langages faciles d'accès qui n'ajoutent pas des difficultés importantes de syntaxe et dont la structure est suffisamment universelle pour permettre un passage en douceur à des langages plus élaborés.