Quelques difficultés de l'enseignement de l'algorithmique et de la programmation aux débutants

Pierre Tchounikine
Université Grenoble Alpes

Les travaux de recherche sur l'enseignement de l'algorithmique et de la programmation aux débutants confirment un certain nombre de constats que les enseignants connaissent bien : les élèves et les étudiants ne développent souvent qu'une compréhension rudimentaire des notions enseignées ; les échecs et abandons sont nombreux ; les résultats sont souvent polarisés avec un groupe qui réussit bien et un autre en grande difficulté.

Cet article propose un éclairage sur trois difficultés spécifiques, différentes mais non indépendantes, qui contribuent à cette situation : le nombre de notions abordées et leurs interrelations ; le rôle des modèles mentaux ; et la question du transfert. Ces problèmes se posent tant au niveau des L1 universitaires que de l'enseignement au lycée (j'utiliserai le mot « étudiant » comme un terme générique pour « étudiant », « étudiante » et « élève »). Les éléments de réflexion proposés ici s'appuient sur les travaux de recherche sur l'enseignement de l'informatique et notamment les synthèses présentées dans [2,5,6], mes travaux de recherche personnels et une longue pratique de l'enseignement en L1.

DOI:10.48556/SIF.1024.26.11

Densité et complexité de l'articulation notionnelle

Pour expliquer ce qu'est un programme informatique il faut en montrer un. Le respect des traditions peut amener à commencer par le traditionnel «*Hello World*» (en C, figure 1.a). Le souhait d'en profiter pour présenter aux étudiants un programme plus proche des exercices à venir peut amener à utiliser un exemple comme celui proposé en figure 1.b (en Python).

```c
#include <stdio.h>
int main(void)
{
printf("Hello World! \n");
return 0;
}
```

(a)

```python
def pg1():
  print("Merci de donner votre nom")
  nom=input()
  print("Bonjour", nom)
  rep=input("Vous avez déjà fait de l'informatique oui/non ?")
  if rep=="oui":
    print("Bon, c'est bien, mais attention ...")
  else:
    print("Pas de souci, on part de zéro, mais ...")
  print("Il va falloir bosser !!!")
```

(b)

Fig. 1. Exemples de programmes présentables en début d'enseignement.

Prenons le programme Python. Lorsqu'on le présente à des étudiants débutants ils devinent sans difficultés comment il fonctionne et ne sont pas surpris par ce que produit son exécution (sauf parfois par le fait que le dernier «`print`» s'exécute dans tous les cas).

Relisons maintenant le code en nous demandant : quelles sont les notions nécessaires à la compréhension de ce programme et de son exécution? Il est possible de prendre cette question à différents niveaux de granularité, mais ce qui est sûr c'est que la liste des notions impliquées est longue : programme et fonction; langage, syntaxe, sémantique, interpréteur, exécuteur; mots clés, syntaxe graphique, indentation; instruction (simple, complexe); variable, identificateur, place mémoire, affectation, type, chaîne de caractère, booléens (ainsi que les entiers ou les réels si l'on prend un exemple numérique); expression, égalité; conditionnelle; entrée-sorties, fonctions «`print`» et «`input`» (et donc, possiblement, paramètre, effet de bord, librairie?); et la liste n'est pas exhaustive.

Ce que cet exemple illustre c'est qu'un programme Python basique — du type de ceux dont on pense qu'il est légitime d'attendre des étudiants qu'ils soient capables de le construire eux-mêmes dès les premières semaines d'enseignement — implique un nombre très important de notions. En fait, ce programme Python mobilise une bonne partie des notions abordées dans un cours d'initiation.

Premier point important : cette complexité n'est pas liée à l'exemple choisi, elle est inhérente à l'informatique. Le programme «*Hello World*» implique

également un bon nombre de notions, et les explications que l'on doit fournir aux étudiants pour qu'ils comprennent une instruction comme «`x = 1`» impliquent bien plus que les notions de variable et d'affectation. Il n'est bien évidemment pas nécessaire que les étudiants maîtrisent parfaitement toutes les notions listées ci-dessus pour commencer à programmer mais, que l'on commence l'enseignement en présentant du code ou pas, il est difficile de ne pas les aborder, d'une façon ou d'une autre, dès les premières séances.

Second point important : la plupart des notions impliquées sont fortement interreliées. Quelle que soit la façon dont on présente les choses, l'étudiant débutant doit donc percevoir et articuler de façon concomitante (plutôt que : une à une, et petit à petit) des notions qui, par ailleurs, relèvent de différents registres. Ainsi, les notions de «variable», «mémoire», «type», «affectation» et «évaluation» sont de natures très différentes, mais il est difficile de les aborder indépendamment les unes des autres.

L'une des caractéristiques importantes de l'enseignement de l'informatique à des débutants est donc la densité et la complexité de l'articulation notionnelle. En quelques séances, les étudiants doivent comprendre ou, *a minima*, développer une perception à peu près cohérente de ce qui, représenté sous forme d'une carte conceptuelle, correspond à un graphe de plusieurs dizaines de nœuds densément interconnectés. Le nombre de notions à présenter avant de pouvoir commencer à faire des exercices impose aux enseignants d'adopter un rythme soutenu, et les étudiants débutants se voient quasi-instantanément plongés dans un nouveau monde conceptuel auquel ils doivent donner sens très vite.

Les travaux de recherche montrent que cette caractéristique joue un rôle important dans les difficultés des étudiants et la polarisation des résultats. Nous (humains) apprenons de façon constructive, en accrochant de nouvelles connaissances à celles dont nous disposons déjà et en bousculant, étendant ou corrigeant celles-ci. Assimiler en même temps plusieurs notions, en particulier lorsqu'il est difficile de les percevoir à l'aune de ce que nous connaissons déjà (et qu'il y a de plus des faux amis, la notion de variable, par exemple) est intrinsèquement difficile et déstabilisant.

Les premières séances d'informatique sont donc très différentes de celles d'autres domaines — maths, physique, etc. — dans lesquels les étudiants ont déjà une base notionnelle sur laquelle s'appuyer pour comprendre et assimiler ce qui est vu en cours (domaines où, par ailleurs, les premiers cours reprennent souvent des éléments de l'année précédente, ce qui peut encore accroître la déstabilisation des étudiants face aux premiers cours d'informatique où, dès le départ, les étudiants doivent comprendre de nouvelles notions). Lorsque chaque cours introduit plusieurs notions nouvelles et que leur maîtrise est nécessaire à la compréhension de celles du cours suivant, la moindre difficulté contamine le reste. Par ailleurs, le fait d'apprendre et

réussir crée de la motivation et de la confiance, ce qui facilite les apprentissages suivants, et à l'inverse ne rien comprendre de ce qu'il se passe crée une dynamique négative, surtout quand les autres étudiants ont l'air de trouver cela facile et que les messages d'erreurs et autres bugs amènent à développer des frustrations et des émotions négatives. Il s'enclenche donc des dynamiques positives (si l'on comprend assez rapidement les principes du programme Python de la figure 1, la suite — boucles, conditionnelles imbriquées, etc. — ne pose généralement pas de difficulté spécifique) et négatives (si l'on ne comprend pas très vite, la suite devient impossible). Dit autrement : si le fameux « déclic » (que je reformulerais en « l'étudiant dispose d'une carte conceptuelle suffisamment complète et cohérente pour lui permettre de comprendre à peu près ce qu'il se passe et commencer à réussir quelques tâches ») ne se fait pas rapidement, les étudiants présentant les atouts généraux de la réussite (assidus, motivés, tenaces, sans soucis matériels ou personnels trop importants) peuvent rattraper le coup, mais pour les autres cela devient très vite mission impossible.

Les implications pour la conduite de l'enseignement sont assez directes :

– expliquer aux étudiants cette caractéristique de l'enseignement de l'informatique pour les aider à ne pas paniquer et à comprendre les efforts qu'ils doivent faire (« il faut travailler dès le début » est une consigne générale qui vaut pour tous les enseignements, mais qui prend un sens et une importance spécifiques en informatique) ;
– s'assurer que les étudiants comprennent les notions et ne développent pas des modèles mentaux obérant la suite des enseignements (cf. section suivante) en leur demandant de reformuler les choses avec leurs mots (par exemple, ce qu'est un type ou une structure de contrôle) ou encore de critiquer des bouts de code qui « marchent » mais sont incorrects ou discutables (par exemple des constructions comme « *expression booléenne* `== true` » ou l'utilisation d'un « `return` » qui brise indûment une conditionnelle ou une boucle) : contrairement à beaucoup d'autres domaines les étudiants peuvent utiliser des notions d'une façon qui leur semble satisfaisante ou, en tout cas, leur permet d'obtenir un programme « qui marche », alors qu'en fait ils ne les comprennent pas vraiment ;
– ne pas penser que s'il y a un groupe d'étudiants qui avance bien cela veut dire que les autres ne travaillent pas assez (de façon contre-intuitive, les étudiants ayant rapidement compris les bases peuvent avancer et réussir en travaillant beaucoup moins que certains autres qui se débattent et s'enferrent en essayant de suivre sans maîtriser les concepts de base) ;
– réfléchir avec attention aux choix pédagogiques que l'on opère (choix des notions traitées et des notions omises ou seulement mentionnées, degré de simplification — voire, de correction — auquel on présente les choses) en fonction des étudiants et des objectifs à terme de l'enseignement (par

exemple, insister dès le départ sur la notion de type pour préparer le passage du paradigme impératif au paradigme orienté objet) ;
– garder en tête que des explications qui sont claires et pertinentes lorsque l'on dispose d'une carte conceptuelle cohérente (comme c'est le cas pour l'enseignant) peuvent être incompréhensibles, voire contre-productives, lorsque la carte conceptuelle avec laquelle on les reçoit est clairsemée ou inexacte (comme c'est le cas, à des degrés variables, pour tous les étudiants).

Il est également possible de chercher à gérer ces difficultés en réfléchissant à l'organisation des enseignements et à sa cohérence avec les points évoqués ci-dessus. En effet, la structure d'enseignement standard (enseignement à un groupe d'étudiants, TP et projets en binômes) amène à gérer de façon synchrone des étudiants qui ne peuvent pas avancer au même rythme, et il est difficile d'aider les étudiants qui présentent des difficultés conceptuelles sans abandonner ceux qui ont passé l'obstacle et dont il faut nourrir la motivation.

Pour essayer de gérer ou au moins de limiter cet écueil il est possible de s'inspirer des initiatives visant à l'aborder de façon radicale, par exemple les modèles de type « *mastery learning* » : le programme d'enseignement est découpé en étapes ciblant chacune un petit nombre de notions et de compétences ; pour chaque étape, l'étudiant dispose de ressources lui permettant de travailler à son rythme (par exemple, documents ou vidéos et exercices autocorrigés) ; lorsque l'étudiant se sent prêt il passe un test (par exemple, des exercices de programmation tirés au hasard d'une banque d'exercices) ; si le test démontre une bonne maîtrise des notions et compétences de cette étape il passe à la suivante, et sinon il continue à travailler et s'entraîner à l'étape courante avant de repasser le test (jusqu'au succès, sans pénalité liée au nombre d'essais). Différentes formes de support additionnel peuvent être proposées, par exemple le fait de passer les tests avec des enseignants ou tuteurs (ce qui permet de faire un point avec l'étudiant) ou encore de proposer des séances de questions-réponses apportant à la fois une aide et un rythme de référence. Confer, par exemple, les expériences relatées dans [3,4].

La mise en œuvre de ce type de modèle pose des problèmes organisationnels importants (et probablement rédhibitoires dans la plupart de nos structures d'enseignement), et par ailleurs n'a pas que des avantages. Ainsi, si ce type d'enseignement semble amener les étudiants à développer des connaissances moins parcellaires et moins fragiles, il ouvre la porte à des comportements de procrastination ou d'efforts *a minima* (juste suffisants pour passer le test)[1]. Les principes, propriétés et limites de cette approche

1. Il y a là un point de discussion incident : que faire d'une méthode d'enseignement qui permet aux étudiants qui jouent le jeu de mieux réussir mais qui donne de moins bons résultats que la méthode d'enseignement standard pour les autres, surtout quand ces derniers sont les plus nombreux ?

peuvent cependant être une source d'inspiration pour l'organisation des enseignements (rythme, contrôle continu, etc.) et l'attention aux étudiants au sein de nos structures plus traditionnelles.

Autre source d'inspiration : les travaux de recherche sur la charge cognitive. Les situations de résolution de problème amènent les étudiants à engager des efforts cognitifs liés à la tâche (au fait de comprendre et résoudre le problème) et à l'apprentissage (au fait d'apprendre). Les travaux montrent que, lorsque les efforts liés à la tâche sont trop importants, cela affecte négativement les apprentissages. Ce constat général s'applique directement à nos exercices d'informatique standards (« construire un algorithme ou un programme qui… »), avec le facteur aggravant que la complexité des interrelations entre les notions impliquées augmente fortement la charge cognitive. Faire travailler les élèves sur des exercices corrigés (lecture et analyse de programmes existants) ou sur des exercices d'arrangement et de modification de bouts de code sont des moyens de limiter cet écueil. Il est bien évident que l'on ne devient pas compétent en algorithmique et en programmation sans construire des algorithmes et des programmes : les activités de résolution de problème sont fondamentales. Cela ne veut pas dire qu'il ne faut enseigner que via ce type d'activité.

Rôle des modèles mentaux

Pour réfléchir et résoudre des problèmes nous (humains) avons besoin d'outils psychologiques qui nous aident à structurer nos processus de pensée. Nous développons donc des modèles mentaux de, par exemple, la notion de variable ou de ce qu'il se passe quand on lance l'interpréteur Python sur le code présenté en figure 1. Sans surprise, les modèles mentaux que développent les étudiants ne sont pas toujours corrects ni pertinents, et cela crée des obstacles aux apprentissages.

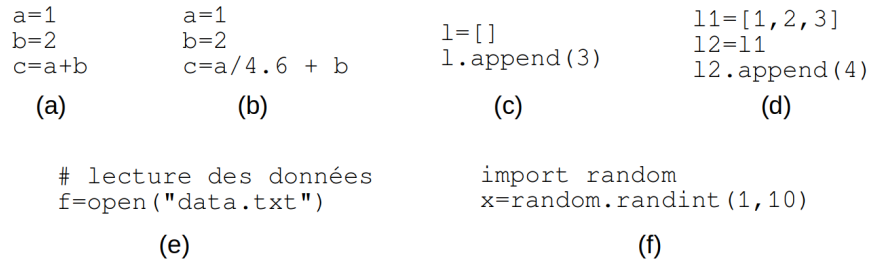

Fig. 2. Affectations en Python.

Prenons les instructions Python présentées en figure 2. Le modèle selon lequel une variable est « une boîte avec une valeur dedans »[2] et une affectation est « le fait de mettre une valeur dans la boîte », modèles que l'on enseigne ou que les étudiants construisent spontanément, permettent de comprendre certains aspects de ce qu'il se passe quand on lance l'interpréteur, mais certains seulement : cela marche bien pour le code (a) ; cela ne rend pas compte du transtypage en (b) ou encore de la première instruction en (c), qui vise moins à « mettre une liste vide dans l » qu'à indiquer que l est une liste et donc permettre les opérations associées aux listes ; cela marche assez bien pour expliquer et comprendre une partie de l'accès partagé créé en (d), mais une partie seulement ; pour (e), ces modèles marchent mal. Les modèles véhiculant l'idée qu'une variable c'est « un identifiant, un type et une valeur », et qu'un type c'est « un ensemble de valeurs et un ensemble d'opérations » permettent de compléter la vision de ce qu'il se passe pour certains de ces cas (mais il y a peu de chances que les étudiants développent spontanément ce type de modèle, d'où l'importance des points d'attention mentionnés précédemment).

De même, les modèles selon lesquels « un ordinateur c'est essentiellement un processeur qui travaille sur de la mémoire » et « chaque ligne d'un programme Python définit une action qui va être appliquée » marchent assez bien. Un étudiant ayant développé cette façon de voir les choses a peu de raisons de la remettre en question : il n'y a pas besoin de déclarer les variables en Python, les instructions que les étudiants sont amenés à lire ou écrire sont très souvent des actions sur des variables, et pour les lignes de type « `if` » ou « `while` » la notion d'action peut être facilement étendue à « passer à un autre bout de code ». Malheureusement, cela va poser des soucis pour comprendre les typages évoqués ci-dessus (une ligne comportant une affectation a en fait plusieurs effets) ou les codes en (e) et (f). Comprendre que, outre la spécification d'une action sur des variables, une ligne de code puisse avoir des finalités aussi différentes que de définir une structure (e.g. une fonction), d'aider les programmeurs qui reliront le programme (ou nourrir le générateur de documentation), permettre la gestion de la mémoire et la vérification de la cohérence des opérations (typage), donner accès au contenu de librairies ou interfacer des opérations liées au système d'exploitation est loin d'être intuitif. Et, bien entendu, expliquer tout cela dès que l'on commence à lire ou écrire des programmes, i.e. dès les premiers cours, n'est probablement pas une bonne idée (cf. section précédente).

Première implication : il est important d'enseigner, très explicitement, des modèles permettant de comprendre ce qu'il se passe à l'exécution. Ne pas le faire (ce qui, d'après les enquêtes, est assez fréquent), c'est-à-dire laisser les

2. La façon de parler des notions informatiques que je reprends ici correspond au type de discours que l'on entend fréquemment dans un cours d'initiation, pas à ce que je propose d'enseigner.

élèves développer un modèle mental idiosyncratique de comment fonctionne l'exécution du code, ouvre la porte à des conceptualisations potentiellement contre-productives. Il est d'autant plus important d'enseigner ces modèles que lorsque nous (humains) avons développé un modèle qui nous est utile, il nous est très difficile d'en changer, et c'est un comportement assez rationnel. Il ne suffit donc pas d'expliquer comment il faut voir les choses, i.e. le modèle dont on pense qu'il va le mieux aider les étudiants à un stade donné d'apprentissage, il faut également identifier et déstabiliser les façons de voir qui sont improductives ou délétères.

Seconde implication : un modèle étant une simplification de la réalité guidée par un but, le but poursuivi et l'intention du modèle (ce qu'il permet de comprendre et de faire) sont des choix d'enseignement majeurs. Comme on l'a vu au début de cet article, l'un des besoins les plus urgents et importants de la plupart des étudiants est de gérer la masse de nouvelles notions (et d'interrelations entre ces notions) des premiers cours. Les modèles proposés aux étudiants devraient donc, *a minima*, prendre en compte ce point.

Ainsi, et pour prendre un exemple non anodin et non consensuel de lien entre les deux écueils évoqués ci-dessus (complexité notionnelle, modèles mentaux), l'initiation à la programmation en Python commence souvent par taper quelques instructions dans la console et regarder ce que cela donne ; l'étape suivante consiste à créer un fichier `.py` dans lequel on écrit un ensemble de lignes effectuant une tâche (e.g. lire les coefficients d'une équation de second degré puis afficher les solutions) et à évaluer le contenu du fichier ; quelques semaines plus tard, on structurera le code en fonctions. La progression semble logique : on part du plus simple et on complexifie. Ceci étant, on prend le risque que les étudiants développent des modèles mentaux inadéquats de ce qu'est un programme ou une fonction (notamment une confusion entre un programme et un fichier contenant des lignes de code), et que cela leur rende plus difficile le passage vers la définition de fonctions ou encore la compréhension des problèmes que posent les variables globales. Dans la mesure où, en toute hypothèse, la notion de programme est (explicitement ou implicitement) présente dès le premier cours, chercher à baisser la complexité notionnelle en écrivant des lignes de code sans les structurer en fonctions ne présente pas que des avantages[3].

3. Autre approche possible : le premier extrait de code montré est une fonction (figure 1.b) et, dès les premiers TD et TP, les étudiants définissent des fonctions et les exécutent en invoquant leur nom (comme ils le font pour lancer leurs applications sur leur smartphone ou leur jeux sur ordinateur). La distinction entre fichier et programme est claire (un fichier peut contenir une ou plusieurs fonctions), et par la suite les notions de procédure et de fonction (au sens de sous-programmes) et de paramètres apparaissent comme un affinement notionnel (et non comme un bouleversement d'une façon de procéder que l'on avait eu du mal à apprendre et « qui marche »).

L'une des spécificités de l'informatique est que les rétroactions de la machine (le compilateur ou l'interpréteur signale les erreurs de syntaxe, l'exécution du programme permet de constater s'il fait ce que l'on avait prévu ou pas) permettent d'aborder certaines parties de l'enseignement de façon « descendante » (typiquement : expliquer une notion puis l'utiliser) mais également, dans une certaine mesure, « ascendante » (typiquement : proposer aux élèves de tester des choses sur machine, puis donner du sens à ce qu'ils ont constaté). Comme les étudiants peuvent créer des programmes sans vraiment comprendre toutes les notions impliquées, ce peut être (et, en tout cas, c'est parfois utilisé comme) une façon de résoudre le problème du nombre de notions à enseigner. Que la pratique soit utilisée comme une application d'un enseignement préalable ou pour amener les étudiants à commencer à comprendre par eux-mêmes via une succession d'essais et erreurs, la centralité des activités de programmation et du couperet de l'exécution par la machine (« ça marche » ou « ça ne marche pas ») a un effet majeur sur les modèles mentaux que développent les étudiants. Comme le fait qu'un programme « marche » ou « ne marche pas » peut être lié à différentes choses, et n'est en tout état de cause pas le point le plus important, vérifier (dès les premiers cours) que les étudiants développent des conceptualisations cohérentes est un enjeu majeur.

Problème du transfert

Le transfert se définit habituellement comme le mécanisme par lequel nous réutilisons nos connaissances dans un nouveau contexte.

En informatique, les phénomènes de transfert vont par exemple jouer un rôle dans la façon dont les étudiants vont aborder leur second langage de programmation. Ainsi, les connaissances acquises dans le contexte de la programmation par agencement de blocs (e.g. en Scratch) peuvent faciliter ou, au contraire, rendre plus difficile, le passage à un langage textuel comme Python. Les conceptions développées via l'usage d'un langage de programmation impératif vont influer sur le passage aux langages fonctionnels ou orientés objets. Etc.

Pour les débutants, le point sur lequel la question du transfert joue un rôle central et majeur relève cependant de l'algorithmique : il s'agit du transfert entre situations de résolutions de problèmes, i.e. de la capacité à se rendre compte que le problème à résoudre présente des similarités avec des problèmes déjà rencontrés, et de s'en servir pour résoudre ce nouveau problème. Pour prendre quelques exemples typiques : se rendre compte que « calculer la moyenne d'une liste de températures » c'est la même chose que le « calculer la moyenne d'une liste de notes » que l'on a traité en cours, et que l'on peut donc utiliser le même schéma algorithmique ; que « chercher le

maximum d'une liste de températures » présente des similarités avec « calculer la moyenne d'une liste de températures » (il faut passer sur tous les éléments, on peut donc utiliser le même type de boucle), mais que le traitement à effectuer est différent ; ou encore que « déterminer s'il y a un 20 dans la liste des notes » nécessite, comme pour « calculer la moyenne d'une liste de notes », de parcourir cette liste (il y a des éléments communs réutilisables donc), mais que l'on n'a pas toujours besoin de parcourir toute la liste et qu'il faut donc faire attention aux conditions d'arrêt.

En tant qu'enseignants nous sommes souvent perplexes, pour ne pas dire désemparés, devant l'incapacité de certains étudiants à voir que le problème proposé est similaire, voire identique à l'habillage près, à un exercice déjà traité. Pourtant, cela correspond tout à fait aux résultats des travaux de recherche sur le transfert. Il a été montré que les humains ont des capacités générales de transfert (mais le fait que l'on puisse les améliorer par apprentissage est une question ouverte). En tant qu'enseignants, nous nous appuyons plus ou moins implicitement sur ces capacités. Cependant, les travaux empiriques montrent que cela marche souvent assez mal, et que les résultats sont généralement très en-deçà de nos espérances (et de nos croyances). Si l'on recentre sur l'informatique, et pour reprendre une polémique ancienne mais toujours très actuelle : l'hypothèse selon laquelle l'algorithmique et la programmation améliorent les capacités de résolution de problème était au cœur des travaux de Seymour Papert sur Logo (dont Scratch est le successeur), les travaux empiriques des années 80 ont mené à la conclusion que ce n'était généralement pas le cas, mais cette hypothèse (positive, sympathique, enthousiasmante, gratifiante) reste cependant solidement ancrée dans la tête de la plupart des enseignants d'informatique et, par exemple, du renouveau des idées de Papert via la notion de « pensée informatique » (*computational thinking*) proposée par Wing [7]. S'il y a certes des expériences positives, la plupart des travaux sur la question montrent malheureusement des résultats modestes, voire inexistants [1]. (Ceci ne doit cependant pas nous conduire à arrêter les travaux sur le sujet, notamment car la question du transfert pose des problèmes méthodologiques qui rendent les travaux extrêmement complexes, ce qui peut et doit nous amener à considérer les résultats actuels, tant positifs que négatifs, avec précaution.)

L'une des raisons qui rendent ce constat (et donc sa prise en compte) difficile est que les mécanismes d'abstraction et d'instanciation et l'utilisation de schémas (au sens large : types de boucle, schémas algorithmiques, design patterns, types de problèmes, frameworks) pour analyser les situations et résoudre les problèmes sont des compétences fondamentales de l'informatique. En tant qu'experts nous « voyons » les similarités (c'est le propre de l'expertise) et nous savons que cette compétence est centrale. Mais, par

définition, les débutants ne sont pas des experts, et il est toujours difficile de savoir ce que « voient » les autres.

Les implications sont assez simples : il ne faut pas penser que multiplier les exemples d'un schéma algorithmique suffit pour que les étudiants en infèrent des structures abstraites pertinentes qu'ils pourront mobiliser pour interpréter et résoudre de nouveaux problèmes similaires ; le transfert spontané marchant très mal, il faut aider les étudiants à détecter les similarités entre problèmes et entre solutions.

La difficulté qui se pose alors est que le fait d'enseigner d'une façon qui favorise le transfert peut entrer en tension avec d'autres considérations. Ainsi, l'enseignement à un niveau abstrait (par exemple, l'enseignement de schémas algorithmiques génériques) favorise le transfert mais, malheureusement, les travaux montrent qu'un enseignement à un niveau abstrait est souvent très difficile pour des débutants (ils n'ont pas assez de connaissances pour relier ces abstractions à des choses qu'ils connaissent déjà et les comprendre vraiment). Inversement, proposer des exercices instanciés sur des cas concrets et des domaines sémantiques familiers des étudiants (par exemple, la gestion de notes) facilite l'apprentissage mais, malheureusement, les détails de surface ont souvent un effet négatif sur les modèles et les schémas de résolution que développent les étudiants. De même, faire travailler les étudiants débutants sur des exercices résolus est très efficace (beaucoup plus que de les laisser sécher sur des tâches de construction d'algorithmes ou de programmes) mais, malheureusement, amène à travailler sur des cas particuliers et ne favorise donc pas l'abstraction.

Il faut donc trouver le difficile et délicat équilibre permettant d'aider les étudiants à aller au-delà des exercices travaillés sans les perdre par trop d'abstraction. Typiquement, proposer à des étudiants débutants d'analyser un exercice comme un problème d'optimisation est sans doute un peu ambitieux. En revanche, analyser une répétition en termes de types de boucle (« Pour », « Tant que ») est devenu une pratique standard (dit autrement : ces structures se sont révélées des abstractions accessibles et utiles). Mettre en évidence des schémas algorithmiques de type « boucle de lecture et vérification de données » (lire puis relire une donnée en boucle tant qu'elle ne respecte pas les critères de validité) ou « application d'un traitement à l'ensemble des éléments d'une liste » est une option possible. Autres exemples : « boucle pour chercher combien » (et la notion d'accumulateur), « boucle pour chercher si » (et les notions de critères et de drapeau), « boucle pour chercher tous les », etc. Quels que soient les choix opérés, les travaux montrent que l'introduction de termes permettant de nommer des buts intermédiaires (ce qu'il faut faire pour mettre en place les schémas algorithmiques enseignés) contribue aux transferts futurs.

Références

[1] Peter J. Denning, and Matti Tedre. 2019. *Computational thinking*. MIT Press.

[2] Mark Guzdial, and Benedict du Boulay. 2019. The History of Computing Education Research. In S. A. Fincher & A. V. Robins (Eds.) *The Cambridge Handbook of Computing Education Research*. Cambridge, UK: Cambridge University Press, p. 11–39.

[3] Brendan McCane, Claudia Ott, Nick Meek, and Anthony V. Robins. 2017. Mastery learning in introductory programming. In *Proceedings of the Nineteenth Australasian Computing Education Conference*, ACM, New York, NY, USA, p. 1–10.

[4] Claudia Ott, Brendan McCane, and Nick Meek. 2021. Mastery Learning in CS1-An Invitation to Procrastinate?: Reflecting on Six Years of Mastery Learning. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V1*, New York, p. 18–24.

[5] Anthony V. Robins. 2019. Novice programmers and introductory programming. In S. A. Fincher & A. V. Robins (Eds.) *The Cambridge Handbook of Computing Education Research*. Cambridge, UK: Cambridge University Press, p. 327–376.

[6] Anthony V. Robins, Lauren E. Margulieux, and Briana B. Morrison. 2019. Cognitive sciences for computing education. In S. A. Fincher & A. V. Robins (Eds.) *The Cambridge Handbook of Computing Education Research*. Cambridge, UK: Cambridge University Press, p. 231–275.

[7] Janet Wing. 2006. Computational thinking. *Communications of the ACM*, 49(3), p. 33–35.