RECHERCHE BINAIRE ET METHODE DE DICHOTOMIE, COMPARAISON ET ENJEUX DIDACTIQUES A L'INTERFACE MATHEMATIQUES - INFORMATIQUE[1]

MEYER[*] Antoine – MODESTE[**] Simon

Résumé – Nous analysons deux problèmes et leurs résolutions qui relèvent du paradigme « diviser pour régner » en algorithmique. Nous montrons que cela permet d'identifier des contenus restés implicites et de mettre en avant des enjeux didactiques à l'interface des mathématiques et de l'informatique.

Mots-clefs : recherche binaire, dichotomie, diviser pour régner, algorithmique

Abstract – We analyse two problems and their solutions, related to the "divide and conquer" algorithmic paradigm. We show that this allows us to identify implicit contents and to highlight didactical issues at the interface between mathematics and computer science.

Keywords: binary search, bisection, divide and conquer, algorithms

I. INTRODUCTION

Dans de nombreux curriculums du secondaire apparaissent des contenus d'algorithmique et de programmation, et plus généralement d'informatique. En France, ces sujets ont été introduits depuis 2009 au lycée (grades 10 à 12) et plus récemment au collège (grades 6 à 9) – voir (MEN, 2009a, 2015, 2017a) pour les programmes du collège et de seconde. Une particularité de ces enseignements est qu'ils ont lieu, entre autres, dans les programmes de mathématiques. Cela questionne la relation entre mathématiques et informatique permise dans ce cadre et pousse à faire des liens entre les deux disciplines. Il faut noter que beaucoup des questions qui se posent à ce sujet dans le secondaire se posent aussi dans l'enseignement supérieur (voir Ouvrier-Buffet et al., 2018).

Par ailleurs, les liens entre mathématiques et informatique sont étroits. On peut mentionner leurs fondements logiques communs, l'apport particulier de l'informatique aux mathématiques et réciproquement les questions que pose l'informatique aux mathématiques, ou encore les champs qui se développent à l'interface des deux disciplines (Modeste, 2015). Le projet ANR DEMaIn[2], dans lequel se développe le travail présenté ici, se propose d'étudier les relations entre mathématiques et informatique en faisant l'hypothèse qu'un travail épistémologique est indispensable pour identifier les

1 Réalisé avec le soutien financier de l'ANR, projet DEMaIn <ANR-16-CE38-0006-01>.
* LIGM (UMR 8049), UPEM, CNRS, ESIEE, ENPC, Université Paris-Est, 77454 Marne-la-Vallée – France – antoine.meyer@u-pem.fr
** IMAG, Université de Montpellier, CNRS, Montpellier – France – simon.modeste@umontpellier.fr
2 Le projet DEMaIn, Didactique et Épistémologie des interactions entre Mathématiques et Informatique, démarré en 2017, a pour objectif d'étudier ces questions particulières et de développer des ingénieries didactiques autour de deux axes complémentaires : Fondements scientifiques (logique, algorithmique, langages, preuve) et Concepts et objets (informatique mathématique, mathématiques discrètes, représentation des objets)

questions didactiques qui émergent de l'introduction d'informatique dans les programmes de mathématiques.

Nous proposons d'étudier deux problèmes classiques, présents dans le secondaire et le supérieur, pour lesquels un algorithme de type *diviser pour régner* existe : la recherche d'élément dans une liste triée et la recherche de zéro d'une fonction. À travers cette analyse, nous souhaitons montrer les questions mathématiques qui sont soulevées, la nature des preuves qui sont en jeu, et la complexité cachée de certaines notions. Ce travail nous semble caractéristique du type de travail qui est à mener autour des questions d'interactions mathématiques-informatique, et met en avant plusieurs points d'articulation : l'étude des algorithmes en tant qu'objets, les rapports entre discret et continu, les notions de complexité et de vitesse de convergence, les enjeux de preuve et de constructivité, les paradigmes algorithmiques[3], etc. La question de la dichotomie nous semble emblématique de ces enjeux.

Nous présentons une étude fine des problèmes concernés pour mettre en avant les enjeux disciplinaires, épistémologiques et didactiques qui se posent. Nous défendrons l'hypothèse que cette analyse préalable permet d'analyser comment certains aspects du thème de la dichotomie sont pris en charge ou non dans les ressources d'enseignement ou de formation.

II. RECHERCHE DE POSITION DANS UNE LISTE TRIEE

La recherche d'élément dans une collection est une tâche essentielle en informatique, à la base de nombreux algorithmes plus complexes. C'est aussi l'un des problèmes classiques abordés dans tout cours universitaire d'algorithmique – voir par exemple (Cormen, Leiserson, Rivest, & Cazin, 1994; Sedgewick, 1988) – et constitue de ce fait un objet d'étude incontournable. Ce problème donne lieu à une résolution emblématique de l'approche diviser pour régner, dont traitera la section suivante. D'un point de vue mathématique, on peut noter que la notion de relation d'ordre, ou d'ensemble ordonné, est au cœur de la question. Il existe plusieurs variantes du problème de recherche de la position d'un élément dans une liste triée. Toutes donnent lieu à des résolutions algorithmiques semblables. Pour les besoins de notre discussion, nous nous arrêtons à la version suivante du problème :

> **Problème P1 : position d'insertion dans une liste triée.** Étant donnés une liste croissante de n entiers $S = (a_i)_{0 \leq i < n}$ et un nombre entier b, déterminer le plus petit indice $0 \leq k < n$ tel que $b \leq a_k$, ou $k = n$ si $b > a_{n-1}$.

La question est de savoir à quel "emplacement" on peut insérer un nouvel élément dans une liste déjà triée tout en préservant sa monotonie. Une réponse k indique que l'élément doit être inséré *entre* les éléments a_{k-1} et a_k (ou en début ou fin de liste si k vaut 0 ou n). On impose aussi que k soit le plus petit possible, c'est à dire que si a_{k-1} existe alors $a_{k-1} < b$. Pour résoudre ce problème, on se place dans un modèle de calcul

3 En informatique, on appelle souvent « paradigmes algorithmiques » les principales familles de méthodes de résolution de problèmes (par exemple l'approche diviser pour régner, la programmation dynamique, les algorithmes de *back-tracking*, etc.)

où les seules opérations élémentaires considérées sont l'accès à un élément de la liste et la comparaisons de deux nombres. On cherche à déterminer un algorithme le plus efficace possible, au sens de la complexité asymptotique en nombre d'opérations élémentaires (*complexité en temps*).

Un algorithme simple (dit de *recherche linéaire*) consiste à balayer entièrement la liste S par indices croissants, en s'arrêtant au premier indice qui convient. Cet algorithme effectue au pire n comparaisons. Une méthode moins coûteuse découle du constat que lorsqu'on compare b avec l'un des a_i, si $b \leq a$ il n'est donc pas nécessaire de considérer les indices supérieurs ; de même si $b > a$ on peut ignorer les indices inférieurs ou égaux à i. Une méthode efficace consiste donc à éliminer le plus grand nombre possible de cas en comparant b à l'élément d'indice médian dans l'intervalle restant à explorer. L'algorithme correspondant est appelé recherche par dichotomie, ou recherche binaire (*binary search*) :

Algorithme A1 : Recherche de l'indice d'insertion de b dans $S = (a_i)_{0 \leq i < n}$.

1. On pose $g = 0$, $d = n$ les bornes initiales de l'intervalle de réponses possibles.
2. Si $g = d$, il reste une seule solution possible, répondre g.
3. Sinon, soit $m = \lfloor (g + d)/2 \rfloor$:
   - Si b $\leq a_m$, on pose $d = m$ et on reprend au point 2 ;
   - Sinon, on pose $g = m + 1$ et on reprend au point 2.

Voici une implémentation itérative possible de l'algorithme A1 en langage Python 3 :

```
def indice_insertion(lst, b):
    g, d = 0, len(lst)    # intervalle initial de recherche
    while g < d:
        m = (g + d) // 2  # indice médian (arrondi par défaut)
        if b <= lst[m]:
            d = m         # on élimine les indices > m
        else:
            g = m+1       # b > lst[m], on élimine les indices <= m
    return g              # ici g == d
```

Nous fournissons ici une analyse détaillée de l'algorithme A1, dans le but d'illustrer le type de techniques mathématiques utilisées pour cela. Nous nous intéressons à la preuve de terminaison de l'algorithme (le fait qu'il s'arrête pour toute instance) ; de sa correction (le fait que la réponse fournie est la bonne pour toute instance), à l'étude de sa complexité (nombre d'opérations nécessaires en fonction de la taille de la liste) et à son optimalité (la preuve qu'il n'existe pas d'algorithme résolvant ce problème avec une meilleure complexité au pire).

Preuve de terminaison. Considérons les valeurs de g et d à chaque exécution de l'étape 2 de l'algorithme A1 sur une instance $(S = (a_i)_{0 \leq i < n}, b)$ donnée. Les valeurs successives de $d - g$ forment une suite strictement décroissante d'entiers naturels, qui atteint donc nécessairement la valeur 0, ce qui provoque l'arrêt de l'algorithme. En effet, en supposant $g \neq d$ à un passage à l'étape 2 (le cas où $g = d$ est trivial), notons g' et d' les nouvelles valeurs de g et d calculées à l'étape 3. Comme $g < d$ on a $g \leq m < d$.

Selon la valeur de a_m on a soit $g' = m + 1 > g$ et $d' = d$, soit $g' = g$ et $d' = m < d$, ce qui implique que$0 \leq d' - g' < d - g$.

Preuve de correction. Soit $(S = (a_i)_{0 \leq i < n}, b)$ une instance du problème, et k le résultat associé. Soit ℓ le nombre de passages par l'étape 2 de l'algorithme (qui est fini comme démontré ci-dessus). Posons $(g_i)_{0 \leq i \leq \ell}$ et $(d_i)_{0 \leq i \leq \ell}$ les suites de valeurs respectives de g et d à chaque exécution de cette étape. Nous allons établir par récurrence sur i la propriété $P(i) : g_i \leq k \leq d_i$. Premièrement, $P(0)$ est vraie car $g_0 = 0$, $d_0 = n$ et par définition $0 \leq k \leq n$. Supposons à présent $P(i)$ pour $0 \leq i < \ell$ et montrons $P(i + 1)$. Par hypothèse de récurrence, $g_i \leq k \leq d_i$. On calcule $m = \lfloor (g_i + d_i)/2 \rfloor$. Deux cas se présentent : si $b \leq a_m$, comme S est croissante et par définition de k on a $k \leq m$. Or $d_{i+1} = m$ et $g_{i+1} = g$, donc $g_{i+1} = g \leq k \leq m = d_{i+1}$. Pour les mêmes raisons si $b > a_m$, nécessairement $k > m$. Or $g_{i+1} = m + 1$ et $d_{i+1} = d$, donc $g_{i+1} = m + 1 \leq k \leq d_i = d_{i+1}$. On déduit de ce qui précède que $P(i)$ est vraie pour tout $0 \leq i \leq \ell$, et en particulier que $g_\ell \leq k \leq d_\ell$, ce qui conclut la preuve puisque $g_\ell = d_\ell$.

Analyse de complexité. Montrons que l'exécution de A sur toute instance (S, b), avec S de longueur n, effectue dans tous les cas $\log_2 n$ opérations élémentaires, arrondi par défaut ou par excès. Plaçons-nous à l'étape 2 de l'algorithme, pour des valeurs quelconques de g et d telles que $0 \leq g \leq d \leq n$. Soit $x = \log_2(d - g + 1)$, nous allons montrer par récurrence forte sur $d - g$ que A1 fournit un résultat en un nombre d'étapes ℓ compris entre $\lfloor x \rfloor$ et $\lceil x \rceil$. Si $d = g$, on vérifie bien que $\ell = \log_2 1 = 0$. Sinon, supposons la propriété vraie à tout rang strictement inférieur à $d - g$. Comme $g \neq d$ la condition de l'étape 2 n'est pas vérifiée, on effectue donc l'étape 3 puis à nouveau l'étape 2. Appelons g' et d' les valeurs de g et d au passage suivant par l'étape 2. On distingue deux cas. Si $d - g + 1 = 2^p$ pour un certain $p \geq 0$, un simple calcul montre que $d' - g' + 1 = 2^{p-1}$. Par hypothèse de récurrence, on obtient que $\lfloor \log_2(2^{p-1}) \rfloor \leq \ell - 1 \leq \lceil \log_2(2^{p-1}) \rceil$, autrement dit $l = p$. Sinon, soit p l'unique entier tel que $2^{p-1} < d - g + 1 < 2^p$. On peut montrer que $(d - g)/2 \leq d' - g' + 1 \leq (d - g)/2 + 1$, ce qui implique que $2^{p-2} \leq d' - g' + 1 \leq 2^{p-1}$. Par hypothèse de récurrence, on obtient donc que $\lfloor \log_2(2^{p-2}) \rfloor \leq \ell - 1 \leq \lceil \log_2(2^{p-1}) \rceil$, soit $p - 1 \leq \ell \leq p$. La propriété est donc vraie pour toutes valeurs initiales $0 \leq g \leq d \leq n$. En particulier pour $g = 0$ et $d = n$ on obtient que l'exécution complète de A1 effectue au pire $\lceil \log_2(n + 1) \rceil$accès aux éléments de S, et exactement p si $n = 2^p - 1$ pour un certain p.

Preuve d'optimalité. On raisonne maintenant sur la classe de *tous* les algorithmes qui résolvent le problème P1 dans le modèle de calcul précédemment décrit. L'ensemble des exécutions de tout algorithme de ce type sur une suite S à n éléments peut être représenté par un arbre binaire appelé *arbre de comparaisons*, dont chaque nœud interne correspond à la comparaison d'un élément de S et d'un autre nombre, et dont chaque feuille représente une réponse de l'algorithme. Il est important de comprendre ici qu'un même arbre représente les exécutions possibles de l'algorithme sur *toutes* les suites de taille n, chaque branche représentant la suite de comparaisons effectuée sur une instance

particulière. La longueur des plus longues branches correspond donc à la complexité au pire sur les suites de cette taille.

On appelle *hauteur* d'un arbre la longueur (le nombre d'arêtes) de sa branche la plus longue. Tout arbre binaire possédant m feuilles est nécessairement de hauteur au moins $\lceil \log_2 m \rceil$, ou dit autrement un arbre binaire de hauteur h possède au plus 2^h feuilles. Dans le cas présent, comme tous les entiers k entre 0 et n sont une issue possible du calcul (positions d'insertion possibles d'un élément b dans une suite S de taille n), chacun d'entre eux doit être représenté par au moins une feuille de l'arbre, qui doit donc avoir au moins $n+1$ feuilles. Ainsi, au moins une des branches doit être de longueur supérieure ou égale à $\lceil \log_2(n+1) \rceil$. On en déduit que *tout* algorithme résolvant ce problème *doit* effectuer au moins $\lceil \log_2(n+1) \rceil$ comparaisons dans le pire cas. On a bien ici un minorant de la complexité du *problème* en lui-même, et non d'un algorithme spécifique le résolvant, puisqu'on a raisonné sur un algorithme quelconque. Puisque l'algorithme A1 étudié précédemment effectue au plus $\lceil \log_2(n+1) \rceil$ accès aux éléments de S (et donc comparaisons), il est dit *optimal*. C'est une caractéristique fondamentale de cet algorithme et de ses variantes.

III. UN CADRE GENERAL : L'APPROCHE "DIVISER POUR REGNER"

L'algorithme de recherche binaire est un représentant de la méthode de résolution de problèmes « diviser pour régner » (*divide and conquer*). La particularité de ces méthodes est d'exhiber une structure commune, décrite par exemple dans (Cormen et al., 1994) :

- Étape **diviser** : on subdivise l'instance donnée en un certain nombre de sous-instances plus petites *du même problème* ;
- Étape **régner**, ou en anglais *conquérir* : on résout récursivement chaque sous-instance, ou si certaines sont de très petite taille on donne leur solution directement ;
- Étape **combiner** : on déduit de la solution à chaque sous-instance une solution à l'instance de départ.

De nombreux problèmes admettent une résolution de ce type, qui permet d'exprimer la complexité des algorithmes-solutions par une formule de récurrence. Dans le cas particulier où l'instance à traiter est subdivisée en sous-instances de tailles (quasiment) égales, un résultat appelé "théorème maître" donne une expression approchée de la complexité :

> **Théorème maître (Cormen et al., 1994, théorème 4.1, p. 70).** Soient $a \geq 1$ et $b \geq 1$ deux constantes, $f(n)$ une fonction, et soit $T(n)$ la suite d'entiers naturels définie par $T(n) = aT(n/b) + f(n)$, où l'on interprète n/b soit comme $\lfloor n/b \rfloor$, soit comme $\lceil n/b \rceil$. $T(n)$ admet l'une des estimations asymptotiques suivantes :
>
> - Si $f(n) = O(n^{\log_b a - \epsilon})$ pour une constante $\epsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.
> - Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log_2 n)$.

- Si $f(n) = \Omega(n^{\log_b a+\epsilon})$ pour une constante $\epsilon > 0$, et si $af(n/b) \leq cf(n)$ pour une constante $c < 1$ et pour n suffisamment grand, alors $T(n) = \Theta(f(n))$.

Les cas de base de la définition de $T(n)$ sont omis de l'énoncé par simplicité. Nous ne présentons pas la preuve du théorème ici mais il est intéressant de noter que celle-ci utilise des arguments du même type que ceux rencontrés ci-dessus, en particulier une réflexion sur l'arbre des sous-instances récursives, sa hauteur et son nombre de feuilles.

Nous disposons ainsi d'une méthode générale pour obtenir une estimation asymptotique de la complexité (disons $T(n)$) de l'algorithme A sur une liste de taille n. En effet celle-ci s'exprime par la formule $T(n) = T(n/2) + 1$, on peut donc appliquer le théorème maître avec $a = 1$, $b = 2$ et $f(n) = 1$. Comme $f(n) = 1 = \Theta(n^0)$, le cas 2 du théorème s'applique et on obtient directement $T(n) = \Theta(n^{\log_b a} \log_2 n) = \Theta(\log_2 n)$.

IV. DEMONSTRATION PAR DICHOTOMIE : L'EXEMPLE DU THEOREME DES VALEURS INTERMEDIAIRES

Le raisonnement par dichotomie est un outil classique des mathématiques, qui apparaît dans la démonstration de théorèmes bien connus d'analyse, comme le Théorème des valeurs intermédiaires ou encore le Théorème de Bolzano-Weierstrass. Il relève d'une approche de type "diviser pour régner", dans un sens moins algorithmique que précédemment évoqué. Ce type d'argument apparaît également sous une forme plus simple dans le raisonnement par disjonction de cas (par exemple selon le signe ou la parité d'un nombre), ou de manière légèrement différente dans le raisonnement par induction structurelle, très utilisé en informatique théorique par exemple. Dans cette section, nous nous intéressons à l'usage d'un raisonnement de ce type dans une démonstration d'un cas particulier du théorème des valeurs intermédiaires, parfois appelé Théorème de Bolzano (Guillemot, 1990) :

> **Théorème des valeurs intermédiaires (cas particulier, dit Théorème de Bolzano).** Pour toute fonction f continue sur l'intervalle $[\![a, b]\!]$ et telle que $f(a)f(b) \leq 0$, il existe $c \in [\![a, b]\!]$ tel que $f(c) = 0$.

Nous proposons ici une démonstration par dichotomie utilisant le théorème (ou axiome) des suites adjacentes. Il est intéressant de noter que la démonstration originelle de Bolzano utilise la propriété de la borne supérieure (Guillemot, 1990), qui peut être vue (à nouveau par un argument de dichotomie) comme une conséquence du théorème des suites adjacentes.

Démonstration – tirée de (Perrin 2017): Si $f(a)$ ou $f(b)$ est nul le résultat est évident. Sinon, on peut supposer $f(a) < 0$ et $f(b) > 0$, quitte à appliquer le théorème à $-f$. On construit par récurrence deux suites adjacentes (a_n) et (b_n), avec $a_0 = a$ et $b_0 = b$. On passe de n à $n + 1$ en considérant $f\left((a_n + b_n)/2\right)$. S'il est positif, on pose $a_{n+1} = a_n$ et $b_{n+1} = (a_n + b_n)/2$, sinon on pose $a_{n+1} = (a_n + b_n)/2$ et $b_{n+1} = b_n$. Soit c la limite commune de (a_n) et (b_n). Comme f est continue, les suites $(f(a_n))$ et $(f(b_n))$ convergent vers $f(c)$. Mais comme $f(a_n) \leq 0$ et $f(b_n) \geq 0$, $f(c) = 0$.

V. DICHOTOMIE EN ANALYSE NUMERIQUE : RECHERCHE DE ZERO

Une application directe de la preuve donnée dans la section précédente consiste à rechercher par une méthode de dichotomie (ou *bisection method*) un encadrement d'un zéro d'une fonction supposée continue et dont le signe change sur l'intervalle étudié. Cette application est exploitée dans de nombreuses ressources à destination des enseignants de lycée, par exemple dans les documents ressources officiels sur l'algorithmique de 2009 et 2017 (MEN, 2009b, 2017b). Ces ressources appellent à une analyse plus détaillée, que nous ne pouvons développer ici faute de place mais sur laquelle nous reviendrons lors du colloque. On considère donc le problème suivant :

> **Problème P2 : encadrement du zéro d'une fonction.** Soit f une fonction continue sur un intervalle I, soient $a < b$ deux points de I et $\epsilon > 0$ un réel. Déterminer $c \in I$ tel que f admet un zéro dans $[c - \epsilon, c + \epsilon]$.

Une fonction Python proposée en réponse à ce problème est fournie dans (MEN, 2017b) :

```
def dichotomie(f, a, b, epsilon=0.0001):
    assert(f(a) * f(b) < 0 and a < b)
    while b - a > epsilon:
        c = (a + b) / 2
        if f(a) * f(c) <= 0:
            a, b = a, c
        else:
            a, b = c, b
    return (a + b) / 2
```

La seconde ligne de cette fonction a pour effet de stopper le calcul si la fonction ne change pas de signe sur l'intervalle $[a, b]$ (ou si l'intervalle est vide). On note cependant que la continuité de f est laissée à l'état d'hypothèse et n'est pas vérifiée faute de moyen approprié. Il est également courant de « protéger » le calcul en imposant une borne fixe sur le nombre maximal d'itérations.

L'une des difficultés rencontrées par ce type de procédure est qu'un programme ne manipule pas des nombres réels mais des représentations approchées (en général des nombres à virgule flottante, de précision limitée). Il est donc concevable que pour certaines fonctions f possédant des points d'images très proches de 0, le résultat renvoyé par la fonction `dichotomie` soit en réalité très éloigné d'un véritable zéro de la fonction, la comparaison `if f(a) * f(c) <= 0` pouvant se révéler trop imprécise. Par exemple, définissons en Python la fonction $f(x) = (x - 0,3).e^{-1/(x-0,3)^2}$, complétée en $0,3$ par $f(0,3) = 0$:

```
def f(x):
    if x == 0.3:
        return 0
    else:
        return (x - 0.3) * exp(-1 / (x - 0.3)**2)
```

Cette fonction est continue et strictement monotone sur et s'annule en $0,3$. L'exécution sur `f` de la fonction `dichotomie` définie ci-dessus donne un résultat incorrect:

```
>>> dichotomie(f, 0, 1)
0.250030517578125
```

On trouve dans (Burden & Faires, 2010) une description détaillée de la méthode de bissection, de ses propriétés (notamment sa vitesse de convergence, qualifiée de « comparativement lente ») et de ses autres limitations (sensibilité aux erreurs d'arrondi), ainsi qu'une discussion sur d'autres critères de continuation possibles à la place du test `b - a > epsilon`. On trouve également dans (Bridges & Vîţă, 2006, p.2) une discussion sur l'aspect *non constructif* de cette méthode, qui implique qu'il n'est pas possible de la « réparer » en recourant par exemple à des représentations des décimaux en précision arbitraire. Leur validité n'étant pas garantie, on peut avancer que les programmes de ce type ne constituent pas des algorithmes au sens strict (mais plutôt des méthodes numériques approchées).

VI. BISSECTION ET RECHERCHE BINAIRE : UNE COMPARAISON

Les procédures de résolution des problèmes P1 et P2 semblent proches et relèvent toutes deux du paradigme diviser pour régner. Mais elles comportent des différences fondamentales, *a fortiori* dans un contexte d'enseignement. Nous citons ici les points les plus marquants.

La première différence entre ces problèmes est leur contexte d'application. Le problème P1 met en jeu des objets discrets, élémentaires en informatique, mais qui ne semblent pas toujours identifiés par les enseignants comme relevant du domaine mathématique qu'ils sont chargés d'enseigner. Le problème P2 met en jeu des fonctions à variable réelle, qui ont un lien fort avec les programmes d'analyse du lycée (notamment le théorème des valeurs intermédiaires), mais peut de ce fait présenter une difficulté mathématique accrue pour les élèves, au risque de masquer les enjeux algorithmiques sous-jacents. En particulier, les questions de choix d'un critère d'arrêt et d'interprétation du résultat obtenu sont des points délicats.

Une autre différence concerne la discussion sur l'efficacité de la procédure. Dans le premier cas, on a vu que la recherche binaire (algorithme A1) est optimale au sens de la complexité algorithmique, grâce à un argument de dénombrement. Par contraste, la notion de vitesse de convergence propre à l'analyse numérique diffère sensiblement de celle de complexité algorithmique « classique », et l'analyse de l'efficacité de la méthode de bissection ne peut être aisément abordée par des arguments du même type. En présence d'hypothèses plus fortes de régularité des fonctions, d'autres méthodes peuvent converger plus rapidement.

Enfin, comme évoqué plus haut, une implémentation directe de la méthode de bissection est sensible aux problèmes d'arrondi liés au choix de représentation des nombres réels. Cet aspect constitue une troisième et dernière différence, peut-être la plus importante dans un contexte d'enseignement.

Il est tentant pour mesurer la distance conceptuelle qui sépare ces deux problèmes de chercher à déterminer, étant donnée une instance (S, b) du problème P1, une instance (f, a, b, ε) du problème P2 dont la solution permettrait d'en déduire la réponse au

problème P1 sur (S, b), et réciproquement (on parle de *réductions*). Cependant, cette tâche s'avère assez fastidieuse et ne semble pas apporter d'éclairage nouveau sur les questions qui nous occupent.

Malgré les différences relevées entre les problèmes P1 et P2, on constate qu'une variante simplifiée du problème P1 (« jeu du nombre à deviner ») est fréquemment convoquée comme préalable (voire comme équivalent simplifié) au problème P2, par exemple dans (MEN, 2009b). Au-delà d'une exposition à l'idée de dichotomie, il ne nous semble pas évident *a priori* que ceci favorise une compréhension fine du problème P2 par les élèves. En particulier, comme discuté précédemment, l'optimalité de l'algorithme de recherche binaire ne peut constituer une justification de la pertinence de la méthode de bissection dans la recherche de zéro d'une fonction, comme le propose MEN (2009b). Le problème P1 général est quant à lui très rarement rencontré, à part dans certaines ressources « expertes » pour la formation d'enseignants. Notre hypothèse est que les problèmes de traitement de données (tri, recherches, etc.) ne sont pas reconnus par les enseignants de mathématiques ou les institutions d'enseignement des mathématiques comme relevant d'une interaction avec les mathématiques (alors que, par exemple, la notion d'ordre y est prégnante).

VII. CONCLUSION ET PERSPECTIVES DIDACTIQUES

Nous avons exposé deux problèmes classiques et leurs solutions, dans deux contextes, l'un en algorithmique classique sur les listes finies, l'autre en analyse numérique approchée. L'approche dominante au lycée du thème de la dichotomie semble être la recherche de zéro d'une fonction, ou plus généralement de mise en œuvre pratique du théorème des valeurs intermédiaires, dans l'objectif éventuel d'en simplifier la compréhension par les élèves. Ainsi, dans (MEN, 2017b), on lit :

> La méthode de dichotomie constitue un procédé dont la compréhension et la mise en œuvre peuvent être particulièrement délicates pour les élèves. Le détour par l'algorithmique permet de « faire fonctionner » la méthode et de l'observer en acte.

On se situe ici dans une démarche d'explicitation des contenus mathématiques par la programmation, sans regard porté sur le programme ou l'algorithme lui-même en tant qu'objet d'étude – cela va dans le sens des observations de Modeste (2012). Nous avons vu que les algorithmes et programmes concernés sont loin d'être simples, en particulier en raison de la complexité du calcul sur les nombres flottants. Enfin, on peut s'interroger sur la compréhension des notions d'algorithmique par les élèves en cas de difficulté sur les notions d'analyse mises en jeu.

Le problème de recherche dans une liste triée et l'algorithme de recherche binaire, tous deux fondamentaux en informatique, sont peu voire pas exploités dans des activités à destination des élèves. Quand ils le sont, c'est souvent sous la forme d'activités d'approche (du type « jeu du nombre à deviner »). Nous émettons l'hypothèse que des activités adaptées sur ce sujet permettraient d'explorer ou de réinvestir plusieurs notions importantes, tant de nature informatique que mathématique, à divers niveaux : notions algorithmiques de base (variables, conditions, boucles), manipulation de listes ou tableaux, découverte progressive de la notion de complexité, approche de l'algorithme en

tant qu'objet, calcul algébrique (puissance, éventuellement logarithme), démonstration (en particulier par récurrence).

RÉFÉRENCES

Burden L. B., Faires J. D. (2010) *Numerical Analysis (9th ed.)*. Brooks & Cole.

Bridges D. S., Vîţă L. S. (2006) *Techniques of Constructive Analysis.* Springer.

Cormen T. H., Leiserson C. E., Rivest R. L., & Cazin X. (1994). *Introduction à l'algorithmique (2ème édition)*. Paris : Dunod.

Guillemot M. (1990) Bolzano et la démonstration du théorème des valeurs intermédiaires. In Barbin E. (Ed.) *La démonstration mathématique dans l'histoire*. IREM de Lyon.

Ministère de l'Éducation Nationale (2009a) Programme de mathématiques, enseignement commun, seconde générale et technologique, BO n°30 du 23/07/2009, http://www.education.gouv.fr/cid28928/mene0913405a.html

– (2009b) Ressources pour la classe de seconde – Algorithmique, http://cache.media.eduscol.education.fr/file/Programmes/17/8/Doc_ress_algo_v25_109178.pdf

– (2015) Programme d'enseignement du cycle 4, BO spécial du 26/11/2015, http://www.education.gouv.fr/pid285/bulletin_officiel.html?cid_bo=94717

– (2017a) Aménagements des programmes d'enseignement de mathématiques et de physique-chimie, seconde générale et technologique, BO n°18 du 04/05/2017, http://www.education.gouv.fr/pid285/bulletin_officiel.html?cid_bo=115984

– (2017b) Ressources pour le lycée - Algorithmique et programmation, http://cache.media.eduscol.education.fr/file/Mathematiques/73/3/Algorithmique_et_programmation_787733.pdf

Modeste S. (2012) *Enseigner l'algorithme pour quoi ? Quelles nouvelles questions pour les mathématiques ? Quels apports pour l'apprentissage de la preuve ?* Manuscrit de thèse. Université de Grenoble. https://tel.archives-ouvertes.fr/tel-00783294

– (2015) Impact of informatics on mathematics and its teaching. On the importance of epistemological analysis to feed didactical research, in Gadducci F., Tavosanis M. (Eds.) *History and Philosophy of Computing*, IFIP Series, Vol.487 (Springer).

Ouvrier-Buffet C., Meyer A., Modeste S. (2018) Discrete Mathematics at University Level. Interfacing Mathematics, Computer Science and Arithmetic. 2nd INDRUM conference, Norway.

Perrin D. (2017) Deux démonstrations par dichotomie. Site personnel. https://www.math.u-psud.fr/~perrin/CAPES/analyse/fonctions/Dichotomies.pdf

Sedgewick R. (1988) *Algorithms.* Addison Wesley.