Chapitre 0 : Algorithmique

Algorithmique (objectifs pour le lycée)

La démarche algorithmique est, depuis les origines, une composante essentielle de l'activité mathématique. Au collège, les élèves ont rencontré des algorithmes (algorithmes opératoires, algorithme des différences, algorithme d'Euclide, algorithmes de construction en géométrie). Ce qui est proposé dans le programme est une formalisation en langage naturel propre à donner lieu à traduction sur une calculatrice ou à l'aide d'un logiciel. Il s'agit de familiariser les élèves avec les grands principes d'organisation d'un algorithme : gestion des entrées-sorties, affectation d'une valeur et mise en forme d'un calcul.

Dans le cadre de cette activité algorithmique, les élèves sont entraînés :
- à décrire certains algorithmes en langage naturel ou dans un langage symbolique ;
- à en réaliser quelques uns à l'aide d'un tableur ou d'un petit programme réalisé sur une calculatrice ou avec un logiciel adapté ;
- à interpréter des algorithmes plus complexes.

Aucun langage, aucun logiciel n'est imposé.

L'algorithmique a une place naturelle dans tous les champs des mathématiques et les problèmes posés doivent être en relation avec les autres parties du programme (fonctions, géométrie, statistiques et probabilité, logique) mais aussi avec les autres disciplines ou la vie courante.

À l'occasion de l'écriture d'algorithmes et de petits programmes, il convient de donner aux élèves de bonnes habitudes de rigueur et de les entraîner aux pratiques systématiques de vérification et de contrôle.

Instructions élémentaires (affectation, calcul, entrée, sortie). Les élèves, dans le cadre d'une résolution de problèmes, doivent être capables : • d'écrire une formule permettant un calcul ; • d'écrire un programme calculant et donnant la valeur d'une fonction ; ainsi que les instructions d'entrées et sorties nécessaires au traitement.
Boucle et itérateur, instruction conditionnelle Les élèves, dans le cadre d'une résolution de problèmes, doivent être capables : • de programmer un calcul itératif, le nombre d'itérations étant donné ; • de programmer une instruction conditionnelle, un calcul itératif, avec une fin de boucle conditionnelle.

Cette partie du cours de seconde n'est donc pas un chapitre individualisé, selon les instructions officielles, mais constitue plutôt une sorte de boîte à outils que l'on va remplir tout au long de l'année, ainsi qu'un recueil des situations algorithmiques que l'on va rencontrer.

1. Notions de base en algorithmique

a) Introduction

Définition : Un algorithme est une succession d'opérations à effectuer sur des données pour atteindre un certain résultat.

Exemples : une recette de cuisine est un algorithme. Les données sont les ingrédients nécessaires et le résultat est le plat que l'on veut cuisiner. En mathématiques, on utilise des algorithmes à tous les niveaux : la détermination du PGCD de 2 nombres, l'exécution d'une simple division, le calcul de l'aire d'un triangle ou celui des décimale de π, etc. relèvent d'algorithmes.

Il faut noter qu'avec les mêmes données de départ un certain algorithme mathématique produira toujours le même résultat. Comme on peut y avoir introduit certains éléments de hasard (le tirage d'un dé par exemple ou l'utilisation d'un nombre pseudo-aléatoire généré par une routine propre au langage informatique utilisé) ceci n'est pas toujours vrai, mais en l'absence d'évènements aléatoires un même algorithme reproduit à chaque fois les mêmes instructions sur le mêmes données ce qui garantit le résultat. Ce n'est donc pas tout-à-fait comme une recette de cuisine qui comporte une foule d'aléas : la taille des œufs varie d'un œuf à l'autre, la durée et la température de cuisson dépendent du four, etc.

On rencontre des algorithmes un peu partout. À chaque fois qu'une opération un peu complexe est réalisée, on la découpe en parties plus simples qui s'exécutent selon une logique propre au résultat souhaité en tenant compte des données disponibles. On va ainsi associer des algorithmes entre eux pour organiser la réalisation d'une tâche complexe. On peut penser à des situations réelles qui demandent une grande

organisation comme un chantier de construction ou l'envoi d'une fusée, mais les algorithmes sont présents dans les moindres détails de la vie de tous les jours. Si je veux sortir de chez moi et que je regarde le temps qu'il fait pour savoir si je dois emporter mon parapluie, j'exécute un algorithme. Pour me laver les dents j'exécute un algorithme. Si mon stylo ne marche pas, je vais vérifier si il contient de l'encre. S'il en reste, j'essaie de presser la cartouche ou je secoue mon stylo ; s'il n'en reste pas, je regarde dans ma trousse s'il y en a une, etc. c'est encore un algorithme.

La description d'un algorithme passe souvent par l'emploi d'un langage codifié, adapté à la situation. Les livres de recettes utilisent souvent une mise en page particulière qui fait ressortir les ingrédients, le temps de préparation, la difficulté. Ils utilisent également un vocabulaire spécifique à la cuisine, qui nécessite des connaissances, c'est-à-dire la connaissance d'opérations qui nécessitent à elles-seules des algorithmes et font souvent l'objet d'un glossaire : séparer le blanc et le jaune d'un œuf, blanchir les oignons, faire un roux, beurrer un moule, etc.

Les algorithmes que l'on va étudier en mathématiques concernent des calculs avec des nombres. Pour les exécuter, il existe différents outils comme le calcul à la main, l'utilisation d'une calculatrice, d'un tableur ou d'un programme informatique. Le programme informatique est la traduction d'un algorithme dans un langage complètement codifié qui va permettre à l'ordinateur de réaliser le travail dans son ensemble. La calculatrice a son propre langage, le tableur a le sien et il y a beaucoup d'autres langages informatiques : Basic, Pascal, Python, C, C++, java, visual Basic, php, javascript, action script pour flash, etc. en sont des exemples plus ou moins spécialisés, adaptés ou non à internet ou à un environnement particulier (contexte graphique, bases de données, ...).

Notre objectif est de concevoir, d'écrire et de corriger des algorithmes puis, éventuellement, de les traduire sous la forme de programmes exécutables. L'exécution d'un programme peut nous inciter à modifier l'algorithme pour généraliser un résultat, ou étendre la recherche. On s'aperçoit parfois seulement à l'exécution qu'un programme « beugue » (de l'anglais bug) car il ne conduit pas au bon résultat ou qu'il « se plante » (en conduisant à une boucle infinie par exemple). Il est bon alors d'exécuter celui-ci pas-à-pas pour examiner le déroulement du traitement. On utilisera à cet effet un environnement de programmation particulier, comme Algobox, qui prévoit cet examen.

L'algorithmique est la science qui étudie les algorithmes (pour atteindre un même objectif, il y a des algorithmes plus ou moins performants..). Nous allons nous initier aux rudiments de cette science, et commencer parallèlement l'étude de certains langages de programmation. La programmation est une technique qui consiste à maitriser à la fois les algorithmes et le langage informatique qui les traduit. Bien entendu, en classe de seconde, on ne peut pas beaucoup s'élever au dessus du niveau très élémentaire en algorithmique bien que certains élèves ont déjà des notions avancées dans certains langages de programmation.

b) Instructions de base

Les nombres que l'on va utiliser sont appelés variables. Il faut dire au départ quels types de variable on utilise : nombres entiers ou nombres à virgule flottante, chaînes de caractères, booléens (variables utilisées en logique qui ne peuvent que prendre les valeurs « vrai » (true) ou « faux » (false), etc. La première instruction sera donc de déclarer les variables que l'on va utiliser. Quand on fait une **déclaration**, on pense à l'ordinateur qui ne sait pas, à priori, ce que l'on veut faire et qui doit réserver une certaine place en mémoire.

Exemple : en pseudo-langage on écrit « A et B sont des entiers », sur algobox « A est-du-type-nombre, B est-du-type-nombre », en java on traduit cela par l'instruction « int A,B; »

Par la suite, il va falloir initialiser le contenu de la variable pour pouvoir l'utiliser. Car au départ, la variable n'est qu'un nom (une adresse en mémoire) auquel ne correspond aucun contenu. L'**initialisation** est une forme de ce qu'on appelle une **affectation**. Affecter la valeur 0 à la variable A se notera souvent A←0 ou aussi parfois 0→A. Dans certain langage, on notera cette affectation A:=0 ou tout simplement A=0. On peut affecter n'importe quel nombre à une variable (pourvu qu'il respecte la forme prévue dans la

déclaration), par exemple A←A+1 est une instruction qui ajoute 1 à la variable A (on ajoute 1 à A et on affecte le résultat à la variable, l'ancienne valeur de A étant écrasée par cette affectation).
Exemple : « A, B et C sont des entiers. A←5, B←3, C←(A+B)×2. », est un algorithme qui calcule le périmètre d'un rectangle de côtés 5 et 3.

Certains algorithmes demandent que l'on introduisent un ou plusieurs nombres dans le processus. Un programme qui calcule le périmètre d'un rectangle mesurant 3 sur 5 n'est pas très intéressant... On préfèrera un programme qui calcule le périmètre d'un rectangle mesurant A sur B, où A et B sont des nombres que l'on introduit au moment de l'exécution. Cette instruction s'appelle une **lecture** : on écrira « Lire la valeur de A » ou « Lire A » et dans le programme cela se traduira de diverses façons, par exemple ?→A (calculatrices Casio), *input* A (calculatrices TI), *enter* A ou *read* A (les langages informatiques utilisent généralement l'anglais). La lecture correspond à une entrée dans l'algorithme, sans préjuger du moyen utilisé : il peut s'agir d'une saisie au clavier, ou bien d'une lecture dans un fichier ou dans une table.

Si des données sont souvent introduites en entrée d'un algorithme par l'utilisateur, des résultats (la production de l'algorithme) sont presque toujours attendus à la sortie. Ces résultats sont alors imprimés ou écrits dans un fichier. Une instruction qui permet d'effectuer une sortie de l'algorithme vers l'utilisateur est appelée une **écriture**. On peut noter cette instruction « affichage » ou « enregistrement » pour faire ressortir le moyen utilisé pour l'écriture (écran ou fichier), dans un programme ce sera des mots-clef comme *output*, *write* ou *print* qui seront utilisés.
Exemple : traduisons le programme de calcul proposé au DNB par un algorithme.

a. Choisir un nombre *x*.	A est un entier	Int A;
b. Calculer le carré de ce nombre.	Lire A	read A;
c. Multiplier par 10.	A←A²	A=A*A;
d. Ajouter 25.	A←A×10	A=A*10;
e. Afficher le résultat.	A←A+25	A=A+25;
	Afficher A	print A;

Instruction conditionnelle : Parfois, il est nécessaire d'effectuer un test. Si la réponse au test est « oui » alors on exécute une certaine instruction mais dans le cas contraire on en exécute une autre (ou on ne fait rien). Par exemple, s'il pleut ou s'il menace de pleuvoir je prends mon parapluie, sinon je sors sans ; cet algorithme nécessite un test « le temps est-il à la pluie? » et selon la réponse je prends ou non mon parapluie. Généralement, on note ce genre d'instruction selon la syntaxe suivante : Si (…) alors {…} sinon {…}. On omet la condition sinon {…} quand elle est vide. Les accolades {…} signifient qu'on écrit une instruction (après alors et après sinon) et les parenthèses indiquent qu'on écrit un test (après si). Dans un programme on écrira souvent *If* (…) *then* {…} *else* {…}.
Un **test** est une affirmation qui peut être vraie ou fausse. Par exemple A>0 ou A=B sont des tests. Certains langages utilisent la notation A==B pour tester si A et B sont égaux, réservant la notation A=B pour l'affectation de la valeur de B à la variable A. Dans certaines situations on associe plusieurs conditions avec le mot « et » (pour que A et B soit vraie, il faut que A soit vraie et B soit vraie, dans tous les autres cas A et B est faux) ou avec le mot « ou » (pour que A ou B soit vraie, il faut que A soit vraie ou que B soit vraie, pour que A ou B soit faux il faut que A et B soient faux). Il est préférable de mettre des parenthèses pour bien individualiser les différents tests qui composent une condition complexe, mais on peut aussi utiliser la règle de priorité de « et » sur « ou ». Par exemple, la condition (A>0) et (A=B) peut être notée A>0 et A=B et la condition ((A>0) et (B>0)) ou ((A<0) et (B<0)) peut être simplement notée A>0 et B>0 ou A<0 et B<0 ou plus simplement encore A×B>0, par contre la condition ((A>0) ou (B>0)) et ((A<0) ou (B<0)) doit garder des parenthèses mais peut être notée (A>0 ou B>0) et (A<0 ou B<0). Nous n'entrerons pas plus loin ici dans cette étude des tests qui relève de l'algèbre booléen.
Exemple : On donne trois nombres a, b et c qui mesurent les côtés d'un triangle et l'on souhaite écrire un algorithme qui détermine si le triangle est particulier (isocèle ou rectangle). On a le choix entre écrire une série d'instructions conditionnelles simples ou une seule complexe. L'algorithme pourra ressembler à :

A, B et C sont des entiers. Particulier est un booléen. Particulier = faux.

Lire A, B et C. Si A=B ou A=C ou B=C alors Particulier = vrai. Si $A^2+B^2=C^2$ ou $A^2+C^2=B^2$ ou $B^2+C^2=A^2$ alors Particulier = vrai. Si Particulier = vrai alors écrire « Le triangle est particulier » sinon écrire « le triangle est quelconque ».

Les conditions peuvent être ici résumée en un seul test :

A=B ou A=C ou B=C ou $A^2+B^2=C^2$ ou $A^2+C^2=B^2$ ou $B^2+C^2=A^2$

Boucle conditionnelle : Parfois on voudrait répéter une même série d'instructions tant qu'une certaine condition n'est pas apparue. Un test doit donc être fait avant ou après la série d'instructions dont le résultat conditionne la poursuite de leur exécution. L'écriture de telles boucles est parfois l'occasion d'erreurs difficiles à détecter car dans certaines situations, le résultat du test ne prend jamais la valeur qui arrêterai la boucle, et celle-ci se poursuit jusqu'à l'infini... D'une façon générale une boucle conditionnelle s'écrit : tant que (test) faire {instructions} (le test est exécuté avant le passage dans la boucle) ou faire {instructions} tant que (test) (le test est alors exécuté après le passage dans la boucle). Dans un programme on écrira souvent *While* (…) *Do* {…} ou bien *Do* {…} *While* (…).

Exemple : On veut jouer avec l'ordinateur, lui demandant de choisir un nombre au hasard entre 0 et 100 que l'on va deviner par des essais successifs. L'algorithme peut ressembler à ceci :

« A et Essai sont des nombres entiers. Choisir A entre 0 et 100. Essai=-1. Tant que Essai≠A faire {Lire Essai. Si Essai=A écrire « Bravo! » sinon {si Essai>A écrire « C'est moins, recommencez! » sinon écrire « C'est plus, recommencez! »}}

Il existe différente façon d'écrire cet algorithme, mais tous doivent produire un même résultat, en particulier afficher les commentaires qui permettent de s'approcher de la solution et d'indiquer la fin de la recherche. Il faut vérifier que les test effectués ne conduisent pas à une boucle infinie. Ici, le programme ne se termine pas tant qu'on n'a pas trouvé la bonne solution, mais on peut ajouter un compteur d'essais et un test qui arrête la recherche si le compteur atteint une certaine valeur. On peut aussi afficher le compteur, calculer un score qui tienne compte du temps écoulé, dessiner un scénario visuel au fur-et-à-mesure de la recherche, etc. l'amélioration d'un algorithme peut conduire à des sophistications incroyables.

Boucle avec un itérateur : Certaines boucles (répétitions d'un ensemble d'instructions) sont contrôlées par un nombre, appelé itérateur, qu'on incrémente (augmente) d'une valeur fixée : le pas de l'incrémentation, qui est généralement 1 mais peut être n'importe quel nombre (entier ou non). On peut généralement choisir de réaliser l'incrémentation au début de la boucle ou à la fin, mais il faut éviter d'écrire des algorithmes qui modifient la valeur de l'itérateur de façon trop compliquée (en la conditionnant à un test par exemple). D'une façon générale une boucle avec itérateur s'écrit : Pour X allant de A à B faire {instructions} (la variable X est initialisée à A et prend comme valeur finale B, l'incrémentation de X se fait dans les instructions). Dans un programme on écrira souvent *For* (…) {…} où toutes les indications utiles pour le contrôle de la boucle sont placées à l'intérieur des parenthèses.

Exemple : On veut faire la liste des multiples d'un nombre A pour en écrire la « table de multiplication ». On doit réaliser une boucle avec un itérateur qui va prendre successivement toutes les valeurs entières de 1 à 10. L'affichage doit ressembler à celui d'une table. Cela donne à peu près ceci :

A et I sont des entiers. Lire A. Pour I allant de 1 à 10 Faire {écrire « A×I = » (A×I). I=I+1}	Int A,I; read A; for (I=1;I<=10;i++) println("+A+" × "+ I+" = "+(A×I))

Dans cet exemple, nous avons mis à droite une traduction de l'algorithme dans un langage proche de Java pour montrer que cette boucle à une forme très codifiée *for* (initialisation;test;finalisation) où finalisation désigne une instruction à réaliser à la fin de la boucle. On voit ici aussi que l'instruction d'affichage peut devenir vite compliquée si on veut afficher simultanément du texte et des valeurs de variables. Cela est prévu dans les langages mais demande un apprentissage, par exemple ici, *println* permet d'aller à la ligne après chaque écriture, le texte est écrit entre guillemets et les variables sont concaténées au texte par des +, lorsqu'on effectue une opération sur les variables, on utilise des parenthèses.

c) Fonctions et autres objets utiles en algorithmique

La connaissance de la syntaxe correcte des différentes instructions ne suffit pas à répondre à tous les problèmes qui se posent en algorithmiques. Il est nécessaire, la plupart du temps d'incorporer certains éléments spécifiques pour simplifier les traitements, ou seulement pour les rendre possible. On peut par exemple, écrire un algorithme pour effectuer une division euclidienne mais comme c'est très souvent utile, les différents langages de programmation proposent des **fonctions pré-programmées** qui effectuent ce travail. La plupart des langages disposent des fonctions suivantes :

ABS(A)	Valeur absolue d'un nombre A
ENT(A) ou FLOOR(A)	Partie entière d'un nombre A
A/B	Avec A et B entiers, partie entière du quotient de 2 nombres entiers
A%B ou A(mod B)	Avec A et B entiers, reste dans la division de 2 entiers (A modulo B)
ROUND(A,n)	Arrondi du nombre A avec n chiffres après la virgule
RANDOM()	Nombre aléatoire (pseudo-aléatoire) compris entre 0 et 1
POW(A,n)	Le nombre A à la puissance n
ROOT(A,n)	Racine nème de A, la racine carrée se notant souvent ROOT(A)

Parmi les autres objets utiles, on peut citer **les tableaux** (aussi appelés tables ou matrices). Ce sont des structures qui contiennent plusieurs variables de la même nature que l'on peut énumérer. Par exemple le nom des mois de l'année peut être déclaré dans un tableau à 12 éléments contenant les 12 chaînes de caractères :
mois={« janvier », « février », « mars », « avril », « mai », « juin », « juillet », « aout », « septembre», « octobre », « novembre », « décembre »}
Dans l'algorithme qui suit une telle déclaration de tableau, on pourra retrouver le nom d'un mois connaissant son rang. Ainsi mois[0] correspond à « janvier » et mois[11] correspond à « décembre ». Les indices utilisés dans un tableau commencent en effet la plupart du temps à zéro. Pour connaître le nom du 9ème mois ou du nème mois, on tapera mois[9−1] ou mois[n−1].
Les tableaux peuvent être définis en clair dans le programme, comme ici pour les mois de l'année, mais ils peuvent être aussi bien construits en cours d'exécution ou bien par des opérations de lecture (saisie de l'utilisateur ou bien lecture dans un fichier).
Il est parfois utile de se servir de tableaux à 2 indices. Par exemple, si on veut stocker les notes d'un élève pour les 3 trimestres dans les 10 matières (maths, français, etc.), on va définir un tableau à 2 indices, le premier donnant le trimestre, et le second la matière. Note[0,1] serait la note en français pour le 1er trimestre (noté 0) tandis que note[2,1] serait la note en français pour le 3ème trimestre et note[1,0] serait la note en maths pour le 2ème trimestre. Ce n'est pas forcément très lisible pour un être humain, mais pour un ordinateur c'est parfaitement clair.
Exemple : Voici un premier algorithme de tri qui réordonne les valeurs d'un tableaux pour les mettre dans l'ordre croissant. Par exemple, le tableau {2,5,3,10,4} sera réordonné en {2,3,4,5,10}. La méthode de tri utilisée est appelée « tri par insertion » : on prend le 2ème nombre et on le met à gauche tant qu'il y a un nombre plus grand à gauche (ici ce n'est pas le cas), ensuite on prend le 3ème nombre et on le met à gauche tant qu'il y a un nombre plus grand à gauche (ici on mettra le 3 entre le 2 et le 7), on continue avec le 4ème nombre (ici on laissera en place le 10) et le 5ème (ici on mettra le 4 entre le 3 et le 5). L'algorithme ressemblera à cela :

```
I, J et N sont des entiers. Lire N (la taille du tableau). Pour I allant de 1 à N faire {lire A[I]}.
Pour J allant de 2 à N faire {X=A[J]. I=J−1.
                              Tant que (I>0 et A[I]>X) faire {A[I+1]=A[I]. I=I−1}
                              A[I+1]=X.}
```

Examinons l'exécution de cet algorithme sur le tableau indiqué, pas à pas :
J=2. X=A[2]=5. I=1. (I>0 et A[I]>X) est faux car A[I]=A[1]=2 et que 2<5, on fait A[2]=5.

J=3. X=A[3]=3. I=2. (I>0 et A[I]>X) est vrai car 2>0 et A[2]=5>3, on entre dans la boucle conditionnelle A[3]=A[2]=5. I=1. (I>0 et A[I]>X) est faux car A[I]=A[1]=2 et que 2<3, on fait A[2]=3.
J=4. X=A[4]=10. I=3. (I>0 et A[I]>X) est faux car A[I]=A[3]=5 et que 5<10, on fait A[4]=10.
J=5. X=A[5]=4. I=4. (I>0 et A[I]>X) est vrai car 4>0 et A[4]=10>4, on entre dans la boucle conditionnelle A[5]=A[4]=10. I=3. (I>0 et A[I]>X) est vrai car 3>0 et A[3]=5>4, on reste dans la boucle A[4]=A[3]=5. I=2. (I>0 et A[I]>X) est faux car A[I]=A[2]=3 et que 3<4, on fait A[3]=4.

Sur cet exemple pourtant très simple, on voit que la description des différentes valeurs prisent par les variables (dont les valeurs du tableau) est rapidement fastidieuse. C'est pourtant une étape nécessaire lorsqu'on cherche à comprendre pourquoi un algorithme n'aboutit pas où il devrait.

Pour finir, examinons les algorithmes rencontrés au collège et même avant, à l'école primaire. Ce ne sont pas forcément les plus simples, même si on les étudie tôt.

a) Algorithme opératoire de la division

La technique opératoire de la division mise en place à l'école et continuée au collège relève de l'algorithmique. Prenons un exemple, divisons 125 par 8 : on divise tout d'abord 12 par 8 (et pas 1 qui est trop petit ou 125 qui est trop grand), ce conduit à 1 au quotient et 4 au reste, puis on divise 45 (le 4 auquel on accole le 5) par 8, ce conduit à 5 au quotient et 5 au reste. Pour une division euclidienne on s'arrête là, et on écrit alors le résultat $125=5\times8+5$. Pour une division décimale, on continue en mettant la virgule au quotient et le 0 des dixièmes au reste, on s'arrêtera lorsque le reste sera nul, et c'est bien le cas ici où l'on trouve que $125\div8=5,625$. Dans le cas d'une division qui ne tombe pas juste, on reconnaît la répétition des restes dès que possible. Par exemple, lorsqu'on divise 12 par 7, on trouve comme restes successifs : 5, 1, 3, 2, 6, 4, 5, 1, 3, 2, etc. Dès lors que l'on reconnaît que 5 est un reste qui a déjà été obtenu, on arrête la division et l'on écrit le résultat : $12\div7=1,714285714285...$ ou plus simplement $12\div7=1,\underline{714285}$ (en soulignant la suite de chiffres qui se répète jusqu'à l'infini).

La description du scénario opératoire est un peu longue car la manœuvre n'est pas simple! C'est pourquoi on l'apprend progressivement à l'école d'abord et au collège ensuite. On l'étend ensuite à la division des nombres décimaux (par exemple le division de 12,3 par 0,56) en prenant en compte la position des virgules. Plus tard, à l'université, on pourra employer le même genre d'algorithme pour diviser des expressions littérales qu'on appelle des polynômes entre eux (par exemple, diviser x^3-1 par $x-1$).

Commençons par la division euclidienne. Quelle est la 1ère étape ? Avant d'effectuer une quelconque division, on doit d'abord s'assurer qu'il y a quelque chose à diviser! Le dividende doit être supérieur au diviseur, sinon le quotient est nul. La division de 12 par 13 par exemple, ne demande d'effectuer aucune division : le quotient est nul, le reste est égal à 12. On effectue d'abord un test : si le dividende est supérieur au diviseur alors on entre dans la procédure de division sinon on s'arrête.

Quelle est la dernière étape ? On écrit le résultat. Dans le langage algorithmique on dira qu'on affiche le résultat. De même, pour que les nombres à diviser entrent dans l'algorithme, on dira que l'on lit leur valeur. Lecture et écriture sont les dénominations appropriées pour effectuer des entrées/sorties vers ou hors de l'algorithme

Appelons D le Dividende, d le diviseur, q le quotient et r le reste. On peut résumer ces premières étapes par l'algorithme partiel suivant (il faudra définir la partie entre crochets appelée *procédure*):

- Lire D et d.
- Si $D<d$ alors $q=0$ et $r=D$, sinon [*procédure*].
- Afficher q et r (éventuellement sous la forme « en ligne » : $D=q\times d+r$.

Pour commencer la procédure, il faut choisir parmi les chiffres de D, en commençant par la gauche, suffisamment de chiffres pour former un nombre qui soit compris entre d et 9d (on suppose que l'on effectue la division en base 10). On est certain que c'est possible de trouver un tel nombre car $D\geq d$ dans la procédure. Mais que fait-on en réalité ? Si D=125 on compare d'abord 1 à d, puis 12 à d, puis (si c'était nécessaire) 125 à d. On doit donc connaître le nombre de chiffres composants le nombre D! Déterminer le nombre de chiffres d'un nombre est une opération élémentaire pour un collégien (et même pour un élève de CP) mais cela constitue un véritable algorithme que l'on pourra traiter ailleurs. Supposons que l'on dispose d'une telle méthode de détermination de n, le nombre de chiffres d'un nombre quelconque. On va donc comparer d'abord le chiffre de gauche à d, puis le nombre formé par les 2 premiers chiffres de gauche à d, puis … jusqu'à trouver un nombre qui soit plus grand que d. On est sûr d'en trouver un, au pire ce sera D lui-même. Il y a donc ici une recherche d'un nombre, qui demande d'augmenter successivement le nombre des chiffres pris à D. On dira qu'on incrémente le nombre de chiffres entre 1 et n chiffres possibles. Avec un peu de recul, on comprendra que, finalement, il n'est pas nécessaire de connaître le nombre n, car on sait d'avance que notre recherche aboutira. Par contre, il faudra trouver un moyen de décomposer notre nombre D en chiffres pour recomposer la suite des nombres à comparer successivement à d. Supposons que l'on sache que $D=a_1a_2...a_n$, les a_i étant les n chiffres composant le nombre D, on va donc comparer à d un nombre $D'=a_1a_2...a_k$, où k est un indice qui va prendre successivement les valeurs 1, 2, … jusqu'à n qui est la plus grande valeur possible.