Introduction à l'Algorithmique
TD1

Ce document a pour objectif de vous familiariser avec l'algorithmique. Les tous premiers algorithmes que vous allez exécuter et écrire sont issus de connaissances communes vues lors de vos cours de l'enseignement primaire ou secondaire.
Pour exécuter les algorithmes en mode dit *pas à pas*, pensez à toujours avoir une feuille de brouillon et un stylo pour pouvoir noter le déroulé de l'algorithme à chaque instruction ou série d'instructions.

Définition informelle d'un algorithme [1] : « *procédure de calcul bien définie qui prend en entrée une valeur, ou un ensemble de valeurs, et qui donne en sortie une valeur, ou un ensemble de valeurs. Un algorithme est donc une séquence d'étapes de calcul qui transforment l'entrée en sortie* ».

1 Problèmes, Solutions, et Types de données

Les algorithmes sont donc des étapes successives permettant d'obtenir un résultat. Il s'agit littéralement de *comment* traiter un problème pour pouvoir le résoudre. Néanmoins, cette séquence d'étapes seule ne permet pas de savoir quel problème on souhaite traiter, il faut donc bien indiquer le contexte et l'objectif de l'algorithme. Ainsi, le problème à traiter, le *pourquoi*, est également extrêmement important.

Chercher et comprendre les problèmes rencontrés est donc très important pour pouvoir écrire les algorithmes les plus efficaces. Par exemple, si l'on cherche à trier des pierres selon leur taille à partir de gravas, on utilisera des tamis de plus en plus petits successivement pour récupérer tout d'abord les pierres, puis les cailloux, et en dernier les grains de sable. Dans cet exemple, il était nécessaire de constater que :

1. la taille des pierres était très importante,
2. des outils permettant de laisser passer ou non des pierres d'une certaine taille sont disponibles,
3. on ne s'intéressait finalement pas à d'immenses rochers.

Pour reprendre la comparaison avec les questions : les pierres et leurs tailles correspondent aux réponses du *quoi*. Ainsi :

— Pourquoi / Quel est l'objectif ? [verbe] **Trier** des pierres par taille
— Quoi / Que manipule-t-on ? [noms] Des *pierres* de différentes *tailles*
— Comment ? En utilisant successivement des *tamis* avec des *trous de plus en plus petits*

Ces spécifications seront très importantes lorsque vous serez amenés à écrire des algorithmes : pensez toujours à bien vérifier les *spécifications* du problème avant d'essayer de répondre au problème (il peut arriver qu'en fait il n'y ait aucun problème).

En algorithmique, il existe quelques types fondamentaux permettant de représenter la plupart des informations du monde physique. En combinant ces types, on peut donc représenter quasiment n'importe quel objet existant ou phénomène mesurable (la taille d'une pierre, sa composition chimique, sa dureté, sa brillance et sa forme une fois taillée, etc).

— entier (*integer* en anglais) : il s'agit des entiers relatifs (positifs et négatifs)
— flottant (*float* ou *double* en anglais) : il s'agit des nombres à virgule (attention, ce type a des problèmes de *précision* : on ne peut pas toujours comparer correctement des flottants)
— caractère (*character* en anglais) : il s'agit des lettres ou caractères (à noter que ce type manipule une seule et unique lettre à la fois)
— chaîne de caractères (*string* en anglais) : il s'agit d'une suite de caractères

1. Introduction à l'Algorithmique. 2001 (2^e édition) T.Cormen et al.

1.1 Types de données

entier	**integer**	Entiers relatifs (+ et -)	**42**
flottant	**float** **double**	Nombres à virgule (attention aux comparaisons)	**13.37**
caractère	**char**	Lettres / Caractères (un seul à la fois)	**'b'**
chaîne de caractères	**string**	Suite de caractères	**"lol"**
booléen	**bool**	Valeur booléenne	**True**

En vous aidant du tableau de types, retrouvez les types de données des valeurs suivantes et corrigez éventuellement les erreurs :

1512	
16.64	
"4chan"	
"/b/"	
xyz	
'P0K3M0N'	
"80,86"	

'M'	
"4242"	
'5'	
3	
"9"	
3.	
"D"	

Indiquez le type de chaque valeur. Si une valeur ne correspond à aucun type, indiquez simplement *incorrect* dans la case à droite.

"3 + 4"	
'3 + 4'	
'"'	
"'"	
"'ABCD'"	
'"ABCD"'	
"'AB'CD"	
"12.21"	
13.37	

2 Exécution pas à pas

2.1 Somme des N premiers entiers

1) Afin de bien comprendre comment fonctionne un algorithme, comment l'exécuter, et potentiellement comment le corriger, utilisez cet algorithme calculant la somme des n premiers entiers en l'exécutant à la main et en remplissant le tableau suivant pour $n = 5$.

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n$$

```
algorithme fonction Somme : entier
  parametres locaux
    entier    n
  variables
    entier    i, sum

debut
i ← 1
sum ← 0
tant que (i <= n) faire
  sum ← sum + i
  i ← i + 1
fin tant que
retourne sum
fin algorithme fonction Somme
```

tour	i	sum
État Initial		
1		
2		
3		
4		
5		
6		

Algorithme de la somme des N premiers entiers

2.2 Multiplication égyptienne

Reprenez maintenant l'exemple de la multiplication égyptienne en l'exécutant cette fois-ci à la main en remplissant le tableau suivant.

2) Vous prendrez comme premières valeurs de test : $a = 4$ et $b = 5$.

3) Remplissez un tableau similaire sur un brouillon pour les valeurs $a = 3$ et $b = 13$.

```
algorithme fonction MultEgpytienne : entier
  parametres locaux
    entier    a
    entier    b
  variables
    entier    x, y, z

debut
x ← a
y ← b
z ← 0
tant que (y > 0) faire
  si (y EST IMPAIRE) alors
     z ← z + x
  fin si
  x ← 2 × x
  y ← y ÷ 2
fin tant que
retourne z
fin algorithme fonction MultEgpytienne
```

tour	x	y	z
État Initial			
1			
2			
3			

Algorithme de la multiplication égyptienne

2.3 Division euclidienne

Effectuez maintenant l'algorithme de la division euclidienne. L'algorithme renverra le quotient.

4) Vous prendrez comme premières valeurs de test : $a = 19$ et $b = 3$.

```
algorithme fonction DivEuclideQuotient :
   entier
  parametres locaux
    entier    a
    entier    b
  variables
    entier    x, y

debut
x ← a
y ← 0
tant que (x > 0) faire
  x ← x - b
  y ← y + 1
fin tant que
retourne y
fin algorithme fonction DivEuclideQuotient
```

tour	x	y
État Initial		
1		
2		
3		
4		
5		
6		

Algorithme du quotient de la division euclidienne

5) L'algorithme fonctionne-t-il tel quel ? Si non, quelle modification faut-il apporter pour obtenir le bon quotient ?

6) Quelle variable faut-il renvoyer pour obtenir le reste ?

7) Si le test dans le *tant que* était un >= plutôt qu'un > : quels changements à l'exécution cela produirait-il ?

8) Cet algorithme est incapable de gérer le cas où 0 est fourni en tant que diviseur. Comment pourrait-on corriger cela afin de protéger l'algorithme d'une boucle infinie ?

Dans la plupart des langages de programmation, vous verrez qu'un opérateur **mod** ou **%** existe et est assez fréquemment utilisée. Il s'agit simplement de l'opérateur calculant le reste de la division euclidienne.
42 % 10 = 2 (lorsque l'on divise 42 par 10, le reste est 2)
40 % 10 = 0 (lorsque l'on divise 40 par 10, le reste est 0)
42 % 2 = 0 (lorsque l'on divise 42 par 2, le reste est 0)
9 % 10 = 9 (lorsque l'on divise 9 par 10, le reste est 9)

3 Écriture d'algorithmes simples

Plusieurs opérations qui nous semblent évidentes sont en réalité bien plus complexes à réaliser dans la pratique.
La multiplication égyptienne est un exemple très concret de cela : nous savons multiplier car nous avons appris et compris ce qu'il se passait lors de cette opération, mais sans l'apprentissage, il est difficile de connaitre le résultat d'une multiplication. Tout comme il est difficile de multiplier de tête des nombres à virgules entre eux.
Les opérations simples sont cependant essentielles à l'écriture de programmes et d'algorithmes plus complexes.

3.1 Multiplication classique

9) Maintenant que vous savez lire, exécuter (y compris en mode pas à pas), et corriger un algorithme, écrivez l'algorithme de la multiplication classique à base d'additions ($N \times M$ = N additions de la valeur M) dans le cas de nombres positifs uniquement.

N'hésitez pas à utiliser un exemple général simple pour bien déterminer la boucle à écrire :

$$5 \times 3 = 5 + 5 + 5 = 15$$

On peut donc s'attendre à avoir une accumulation dans une variable pour le résultat :

$$0, 5, 10, 15$$

$$0(+5)$$
$$5(+5)$$
$$10(+5)$$
$$15$$

Mais, on doit également connaître le cas d'arrêt : lorsque le multiplicateur est à 0.

val1	val2	sum
5	3	0
5	2	5
5	1	10
5	0	15

```
algorithme fonction MultClassique : entier
  parametres locaux
    entier    a
    entier    b
  variables
    entier

debut









fin algorithme fonction MultClassique
```


10) Comment peut-on traiter les nombres négatifs ? Écrivez maintenant une fonction *MultRelatifs* permettant de multiplier des nombres entiers négatifs (n'hésitez pas à appeler la fonction de multiplication que vous avez précédemment écrite).

3.2 Puissance

11) Écrivez l'algorithme calculant la puissance de x^n (pour n positif ou nul).

Au lieu d'utiliser les symbole $\times$ ou * pour multiplier, vous ferez un appel à votre dernière fonction *MultRelatifs* en lui donnant deux paramètres et en récupérant le résultat.

```
algorithme fonction Puissance : entier
  parametres locaux
    entier    a
    entier    b
  variables
    entier

debut









fin algorithme fonction Puissance
```


3.3 Parité

12) Écrivez l'algorithme testant la parité d'un nombre n.

La parité est simplement la qualité d'un nombre d'être pair ou impair. Vous renverrez 0 en cas de nombre pair, et 1 en cas de nombre impair.

```
algorithme fonction Parite : entier
  parametres locaux
    entier    n
  variables
    entier

debut









fin algorithme fonction Parite
```

4 Réécriture de boucles

Les boucles *For* et *While* permettent de répéter des opérations un certain nombre de fois.

4.1 Boucle For

La boucle *For* est la plus simple à utiliser : on fixe le nombre de fois que les instructions doivent être réalisées, puis elles sont exécutées. Cette boucle permet, dans certains cas, au *compilateur* d'optimiser l'exécution en recopiant N fois les instructions à exécuter sans tester l'état de l'*itérateur*, ou encore en parallélisant l'exécution des instructions.

Néanmoins, la boucle *For* ne permet pas de tester des conditions spécifiques, mais seulement de répéter des instructions un nombre précis de fois.

Attention, dans certains langages comme le C, la boucle *For* est en réalité une boucle *While* dont l'itérateur est testé à chaque tour de boucle : lorsque vous écrivez une boucle *For*, faites très attention à ne **jamais** modifier l'itérateur dans la boucle au risque de faire une boucle infinie, ou à l'inverse que votre modification ne soit pas prise en compte.

4.2 Boucle While

La boucle *While* teste une condition (ou un ensemble de conditions) et, si la condition est vérifiée, les instructions sont exécutées une fois de plus. Par définition, on ne peut donc pas facilement paralléliser ou optimiser ce genre de boucles : on doit tester à chaque fois si l'on peut continuer l'exécution.

Le but de la boucle *While* est bien évidemment de tester des valeurs modifiées par les instructions contenues dans la boucle elle-même.

4.3 Réécriture de boucles

Transformez les boucles *For* suivantes en boucles *While*.

```
algorithme fonction LoopF1 : entier
  parametres locaux
    entier    a
  variables
    entier    iter, cpt

debut
cpt ← 0
pour iter de 0 à 5 faire
  a ← a / 10
  si (a >= 0) alors
    cpt ← cpt + 1
fin pour
retourner cpt
fin algorithme fonction LoopF1
```

```
algorithme fonction LoopF2 : entier
  parametres locaux
    entier    a
  variables
    entier    iter, res

debut
res ← 0
pour iter de 1 à a faire
  res ← res × a
fin pour
retourner res
fin algorithme fonction LoopF2
```

Transformez les boucles *While* suivantes en boucles *For* si cela est possible, sinon, indiquez *impossible*.

```
algorithme fonction LoopW1 : entier
  parametres locaux
    entier    a
    entier    b
  variables
    entier    res

debut
res ← a
tant que (res ≠ 10) faire
  res ← res + 1
  a ← a + b
fin pour
retourner a
fin algorithme fonction LoopW1
```

```
algorithme fonction LoopW2 : entier
  parametres locaux
    entier    a
    entier    b
  variables
    entier    res

debut
res ← a
tant que (res ≠ 10) faire
  res ← res + 1
  a ← a + b
fin pour
retourner a
fin algorithme fonction LoopW2
```

5 Fonctions, Procédures, et Effets de bord

La programmation, par rapport à l'algorithmique, est l'implémentation concrète des algorithmes dans des ordinateurs. C'est-à-dire que la logique de l'algorithme est développée pour qu'un ordinateur puisse l'exécuter avec son (ou ses processeurs) et périphériques.

5.1 Portée des Variables

Dans les algorithmes que vous écrivez, et leur version implémentée dans un programme, vous avez utilisé des *variables*. Ces variables ne peuvent pas être utilisées avant d'avoir été déclarées. Ainsi, une variable dont on ne connait pas l'existence ne peut pas être utilisée (ni en écriture ni en lecture).

Dans certains langages (comme le PHP ou le Python), on peut déclarer des variables à la volée. Néanmoins, la plupart des langages de programmation exigent que le développeur déclare chaque variable utilisée, et parfois avec le type de données qu'elles contiendront (comme en C). Ceci permet en réalité au compilateur de réserver suffisament d'espace pour la variable avant d'exécuter le programme.

Au cours de l'histoire, il était initialement nécessaire de déclarer les variables utilisées dans l'ensemble d'un programme, puis, il est devenu possible (et plus optimal) de ne déclarer les variables que dans les *scopes* où celles-ci sont utilisées. Ainsi, aujourd'hui, vous pouvez définir des variables dont la durée de vie ne sera limitée qu'à un scope encore plus réduit que celui d'une fonction (par exemple uniquement dans une boucle).

Les paramètres sont des cas spéciaux de variables dont la valeur est fixée lors de l'appel de la fonction, mais les modifier ne changera pas leur état en dehors de la fonction appelée, sauf dans certains cas spéciaux que vous verrez beaucoup plus tard.

```c
int mul_two(int a, int b)
{
  int res = 0;
  int iter = 0;

  while (iter != b)
  {
    int tmp;
    tmp = res + a;
    res = tmp;

    iter = iter + 1;
  }
  return (res);
}
```

Dans cet exemple, les variables *res* et *iter* peuvent être utilisées n'importe où dans la fonction. Mais, la variable *tmp* n'existe que dans la boucle.

Ainsi, *tmp* ne peut ni être retournée, ni même lue en dehors de la boucle. Étant donné que la boucle est répétée plusieurs fois, la variable *tmp* est même supprimée et redéclarée à chaque tour : vous ne devez jamais partir du principe qu'elle contiendra la précédente valeur allouée.

5.2 Fonctions et Procédures

Jusqu'à maintenant, les algorithmes que vous avez écrit prenaient des valeurs en paramètres, et retournaient chacun une valeur : il s'agit en programmation de *fonctions*. Mais il existe également ce que l'on appelle des *procédures*.

— Une *fonction* retourne une valeur
— Une *procédure* ne retourne aucune valeur

5.3 Effets de bord

L'existence des procédures est justifiée par le fait que celles-ci modifient des valeurs en dehors du corps de la procédure : ce ne sont pas juste les variables de la proécdure ou ses paramètres d'entrée qui sont modifiés, mais bien les variables externes à la procédure.

Par exemple, jusqu'à maintenant, on vous demandait de retourner une valeur à la fin d'une fonction (par exemple : calculer la somme des N premiers entiers, et retourner le résultat). Or, vous avez probablement trouvé beaucoup de tutoriaux en ligne vous indiquant comment écrire à l'écran des valeurs (souvent avec une fonction ou procédure appelée *print()*). Écrire à l'écran n'est pas du tout une fonctionnalité triviale : on doit prendre la valeur donnée, la transformer en caractères affichables, placer ces caractères dans une section de la mémoire accessible par la sortie graphique, lire les caractères pour décider quels pixels allumer ou éteindre à l'écran, et surtout, où écrire ces caractères à l'écran.

Tout cela n'implique aucun retour vers l'utilisateur (qui a simplement demandé à afficher quelque chose à l'écran), mais modifie énormément de valeurs dans le système d'exploitation et dans l'ordinateur, c'est-à-dire en dehors du programme initialement écrit.

La modification de valeurs en dehors de la fonction/procédure s'appelle des *effets de bord*.

Un *effet de bord* est donc simplement une modification dans une variable en dehors du scope actuel (en général dans un scope de plus haut niveau).

Les procédures ont comme unique objectif d'effectuer des effets de bords, et donc, de ne rien avoir à retourner comme valeurs. À l'inverse, des fonctions peuvent appliquer des effets de bord *et* retourner une valeur (par exemple pour indiquer combien de caractères ont réussi à être affichés ou transmis), mais ce n'est pas leur unique objectif.

6 Propriétés des fonctions

L'art de la programmation réside en partie dans l'écriture de fonctions possédant certaines propriétés :

- *Déterminisme* : une fonction est dite déterministe si elle renvoie toujours le même résultat lorsqu'on lui donne les mêmes paramètres
- *Pure* : une fonction est dite pure si elle est déterministe *ET* ne produit aucun effet de bord

De nombreuses autres propriétés existent (les fonctions réentrantes, idempotentes, ...) et vous les découvrirez plus tard.

Ce document et ses illustrations ont été réalisés par Fabrice BOISSIER en septembre 2024 (dernière mise à jour octobre 2024)