Bases Fondamentales
TD0 - Exécution

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, on utilisera des tamis de plus en plus large successivement pour récupérer tout d'abord les grains de sable, puis les cailloux les plus petits, et en dernier les pierres de plus grande taille (si l'on utilisait un tamis trop grand dès le début, toutes les tailles passeraient sans distinction). Dans cet exemple, il était nécessaire de constater que la taille des pierres était très importante, que des outils permettent de laisser passer ou non des pierres d'une certaine taille sont disponibles, et que l'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 ? **Trier** des pierres par taille
— Quoi / Que manipule-t-on ? Des pierres de différentes tailles
— Comment ? En utilisant successivement des tamis avec des trous de plus en plus grands

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 tout ce qui existe et est 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.

2 Exécution pas à pas

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
0		
1		
2		
3		
4		
5		
6		

Algorithme de la somme des N premiers entiers

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
0			
1			
2			
3			

Algorithme de la multiplication égyptienne

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
0		
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.

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.

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).

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
```


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
```

Ce document et ses illustrations ont été réalisés par Fabrice BOISSIER en octobre 2022
(dernière mise à jour octobre 2023)