Logique et Récursivité
TD1

Ce document a pour objectif de vous familiariser avec l'algorithmique et l'écriture d'algorithmes. Les algorithmes que vous allez écrire sont issus de connaissances un peu plus avancées en mathématiques ou en sciences de l'ingénieur vues au lycée.
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 Logique

En mathématiques, un domaine étudie en particulier la *logique* de façon formelle (logique de premier ordre, ...). Ce domaine est directement appliqué en électronique numérique avec les portes logiques (*logic gates* en anglais) et les bascules (*flip-flops* en anglais).

La logique s'appuie sur deux valeurs qui s'opposent : **vrai** (1) et **faux** (0). Il est possible d'exprimer des assertions avec des *formules logiques* afin de vérifier si celles-ci sont vraies ou fausses dans certains cas selon des paramètres. Pour cela, plusieurs opérateurs logiques existent. Les trois opérateurs fondamentaux (faisant découler toute une série d'autres opérateurs utiles) existent : **NOT** (non), **AND** (et), **OR** (ou).

NOT est un opérateur unaire, il ne s'applique qu'à un seul paramètre : « - A ».
AND et **OR** sont des opérateurs binaires, ils s'appliquent à deux paramètres : « A et B », « A ou B ».
Nous représentons dans les tableaux suivants ce qui s'appelle des *tables de vérités* : les résultats des assertions logiques.

A	**NOT**
0	1
1	0

A	B	**AND**
0	0	0
0	1	0
1	0	0
1	1	1

A	B	**OR**
0	0	0
0	1	1
1	0	1
1	1	1

Ces trois opérateurs sont très souvent utilisés pour tester des conditions. Si le résultat est vrai (1), alors la condition est validée. On peut combiner ces opérateurs pour tester des assertions logiques.

Cependant, ces opérateurs sont également appliqués dans le cadre de langages de programmation pour modifier des valeurs au niveau des bits les constituant (comme nous le verrons plus tard). En plus de **NOT**, **OR**, **AND**, vous pourrez rencontrer les opérateurs **NAND** (non et), **NOR** (non ou), et le très important **XOR** (ou exclusif) qui est extrêmement utilisé en cryptographie. Ces opérateurs sont simplement des combinaisons des trois opérateurs fondamentaux : **NAND** applique simplement un **NOT** au résultat du **AND**, et **NOR** applique simplement un **NOT** au résultat du **OR**.
XOR est légèrement plus complexe dans sa construction, mais sa logique est simple : il faut que les deux entrées soient dans des état différents pour que le résultat soit vrai (1). Vous pouvez néanmoins constater qu'en appliquant un **AND** aux sorties de **OR** et **NAND**, on obtient la même table de vérité.

A	B	**NAND**
0	0	1
0	1	1
1	0	1
1	1	0

A	B	**NOR**
0	0	1
0	1	0
1	0	0
1	1	0

A	B	**XOR**
0	0	0
0	1	1
1	0	1
1	1	0

2.1 Exercices de logique

1) Écrivez une fonction prenant 3 entiers en paramètre, et indiquant lequel est le plus petit/grand. *min(a, b, c)* *max(a, b, c)*
2) Améliorez ces fonctions pour qu'elles prennent en chargent les cas où des valeurs égalent sont données en paramètres : (3, 2, 2), (3, 4, 3), (7, 7, 7), ...
3) Écrivez une fonction effectuant la somme des carrés des 2 plus grands/petits nombres parmi 3. *SumSquare(a, b, c)*

3 Récursivité

La récursivité est un principe très simple où une fonction se rappelle elle-même.

L'écriture d'algorithmes récursifs implique au moins deux choses dans cet ordre très précis : une condition d'arrêt où l'on retourne le résultat, puis, un appel récursif avec un paramètre modifié (si on ne modifie aucun paramètre, alors la récursion serait infinie : on rappellerait la fonction dans les mêmes conditions qu'actuellement, donc elle se rappellerait encore une fois avec strictement les mêmes paramètres).

Il est nécessaire d'écrire les conditions d'arrêts en premier, car il s'agit des cas exceptionnels où l'on doit arrêter la récursion. De même, il est hautement conseillé d'écrire les cas les plus génériques en dernier, car des cas partiellement exceptionnels pourraient être absorbés plus tôt par le cas général. Par exemple, si l'on souhaite dénombrer les N premiers entiers, et afficher à chaque dizaine un message particulier, on déclarera en premier la condition d'arrêt où un paramètre est à 0 ou 1, puis, on écrira la condition testant les valeurs notables particulières (ici si le paramètre est une dizaine), et en tout dernier on écrira le cas général qui concerne n'importe quelle valeur.

3.1 Exercices récursifs

Maintenant que vous avez écrit quelques algorithmes simples avec des boucles, nous allons passer à leurs versions récursives.

4) Commencez par exécuter l'algorithme de la somme des N premiers entiers en remplissant le tableau avec l'évolution des paramètres donnés dans un premier temps, puis des résultats. Vous effectuerez cette exécution avec 5 comme paramètre.

```
algorithme fonction SommeRec :
   entier
  parametres locaux
    entier    n

debut
si (n == 1) alors
  retourne (1)
sinon
  retourne (n + SommeRec(n - 1))
fin si
fin algorithme fonction SommeRec
```

appel	n
0	
1	
2	
3	
4	
5	
6	

appel	retour	total
6		
5		
4		
3		
2		
1		
0		

Somme des N premiers entiers (récursif)

Vous remarquerez que l'algorithme est beaucoup plus court en quantité d'instructions. Ceci est principalement dû au fait que le calcul que nous exécutons est déjà dans une forme adaptée (souvenez-vous du principe de récurrence, ou encore des suites) : la même opération est répétée avec un paramètre réduit ou augmenté de 1 (ou d'un pas bien défini).

5) Écrivez maintenant l'algorithme de la factorielle, mais de façon récursive. N'oubliez pas : on écrit d'abord la ou les conditions d'arrêt, et ensuite seulement on effectue l'opération avec l'appel récursif.

$$n! = 1 \times 2 \times 3 \times \ldots \times n$$

6) Écrivez l'algorithme récursif calculant la somme des N premiers entiers.

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

7) Écrivez l'algorithme récursif calculant la multiplication de deux entiers positifs, en n'utilisant que des additions et des soustractions.
8) Améliorez l'algorithme de la multiplication pour qu'elle gère maintenant les nombres négatifs. Vous pouvez pour cela vous aider d'une fonction chapeau, c'est-à-dire une fonction qui prend les deux paramètres attendus (les deux nombres à multiplier) et fait différents tests avant d'appeler une autre fonction qui elle sera récursive.

9) Écrivez l'algorithme récursif calculant le Nième terme d'une suite géométrique. Vous devriez avoir en paramètres : le terme u_0 désignant le premier terme de la suite, la raison q, et le numéro n du terme recherché.

$$u_n = u_0 \times q^n$$

10) Écrivez une fonction récursive calculant le $n^{\text{ème}}$ terme de la suite de Fibonacci.

$fibo(0) = 0$
$fibo(1) = 1$ (pour simplifier, on considérera que : $fibo(0) = fibo(1) = 1$)
$fibo(n) = fibo(n-1) + fibo(n-2)$

11) Écrivez maintenant la version itérative de la suite de Fibonacci. [Astuce : on dispose de deux cas à valeurs fixes, et à chaque étape on doit se rappeler du résultat précédent.]

12) Écrivez une fonction récursive calculant le $n^{\text{ème}}$ terme de la suite d'Ackermann.

$$\begin{aligned} A(0,n) &= n+1 && [n >= 0] \\ A(m,0) &= A(m-1,1) && [m > 0] \\ A(m,n) &= A(m-1, A(m,n-1)) && [m > 0 \ \& \ n > 0] \end{aligned}$$

13) Écrivez l'algorithme récursif calculant le nombre de combinaisons de p dans n (C_n^p ou CPN), c'est-à-dire le nombre de parties à p éléments dans un ensemble E contenant n éléments.

Par exemple : pour $p = 2$, on recherche tous les **couples** possibles de différents éléments. Pour $p = 3$, on recherche tous les **triplets** possibles de différents éléments. En indiquant $n = 3$, on vise un ensemble composé de trois éléments distincts (par exemple : $E = \{1, 2, 3\}$, ou $E = \{A, B, C\}$, ou $E = \{\spadesuit, \heartsuit, \clubsuit\}$, il s'agit juste de trois éléments distincts).

Ainsi, pour $p = 2$ et $n = 3$, on recherche tous les couples possibles de trois éléments :

(A,B) (A,C)
(B,C)

$\Rightarrow C_3^2 = 3$ (3 couples possibles)

Pour $p = 2$ et $n = 4$, on recherche tous les couples possibles de quatre éléments :

(A,B) (A,C) (A,D)
(B,C) (B,D)
(C,D)

$\Rightarrow C_4^2 = 6$ (6 couples possibles)

Pour $p = 3$ et $n = 3$, on recherche tous les triplets possibles de trois éléments :

(A,B,C)

$\Rightarrow C_3^3 = 1$ (1 triplet possible)

Pour $p = 3$ et $n = 4$, on recherche tous les triplets possibles de quatre éléments :

(A,B,C) (A,B,D) (A,C,D)
(B,C,D)

$\Rightarrow C_4^3 = 4$ (4 triplets possibles)

Voici les axiomes pour votre implémentation :

$$
\begin{array}{ll}
C(0,n) = 1 & \\
C(n,n) = 1 & [n \neq 0] \\
C(p,n) = C(p,n-1) + C(p-1,n-1) & [n \neq p]
\end{array}
$$

3.2 Exercices variés (récursif & itératif)

Les exercices dans cette section doivent plutôt être réalisés en itératif. Il est précisé lesquels peuvent aisément être réalisés en récursif. Il est interdit d'utiliser les tableaux ou pointeurs pour réaliser ces exercices.

14) Écrivez une fonction transformant un format horaire en un format uniquement composé de secondes. Cette fonction prendra 3 entiers en paramètre (les heures, les minutes, et les secondes) et les convertira en secondes. Par exemple, 1h 23m 45s deviendra 5025 secondes. *ConversionHoraire1(hh, mm, ss)*

15) Écrivez une autre fonction de conversion horaire qui prend cette fois un unique entier qui respecte un format précis (hhmmss) pour le convertir en secondes. Par exemple le paramètre 153042 signifie 15h 30m 42s qu'il faut convertir en secondes. *ConversionHoraire2(hhmmss)*

16) Écrivez une fonction qui transforme un nombre en son miroir. Cette fonction prend un entier en paramètre, et construit un autre entier qui est son miroir. Par exemple, pour 12034, son miroir sera 43021. Autre exemple : 2000 aura comme miroir 0002, c'est-à-dire 2. Attention aux nombres composés d'un nombre pair/impair de chiffres. Commencez par réaliser une fonction itérative. *MiroirIter(n)*

17) Écrivez maintenant la version récursive du nombre miroir. Pour cette première version récursive, vous appelerez une fonction *écrire(x)* ou *print(x)* qui affiche un caractère ou un chiffre à la fois. *MiroirRec1(n)*

18) Écrivez maintenant la version récursive du nombre miroir. Pour cette deuxième version récursive, vous devrez renvoyer le nombre miroir et non pas juste l'afficher. *MiroirRec2(n)*

Astuce : vous pouvez utiliser un *accumulateur* comme deuxième paramètre, donc, écrire une fonction chapeau qui prendra un seul paramètre et préparera l'appel à la fonction récursive.

19) Écrivez une fonction récursive qui affiche les éléments successifs de la conjecture de Syracuse, mais qui renvoie également le nombre d'éléments produits avant d'atteindre 1. Utilisez les fonctions *écrire(x)* ou *print(x)* et une fonction chapeau si nécessaire. *Syracuse(n)*

Voici les axiomes pour votre implémentation :

$$\begin{aligned} &Syracuse(0) = 1 \\ &Syracuse(1) = 1 \\ &Syracuse(n) = n/2 \qquad \text{si } n \text{ est paire} \\ &Syracuse(n) = 3n + 1 \quad \text{si } n \text{ est impaire} \end{aligned}$$

20) Écrivez une fonction détectant si un nombre est un palindrome. La fonction renvoie *vrai* si c'est un palindrome, sinon elle renvoie *faux*. Un palindrome est simplement un mot ou un nombre composé des mêmes caractères ou chiffres sur sa première moitié par rapport à sa deuxième moitié. Par exemple, 27972 est un palindrome. 1331 est également un palindrome, mais 1664 n'en est pas un. Faites d'abord une version itérative, puis faites une version récursive. *PalindromeIter(n)* *PalindromeRec(n)*

4 Récursivité terminale

Lors de l'exécution de certains algorithmes récursifs, vous avez du vous rendre compte qu'avec certains paramètres trop grands, l'ordinateur refuse d'exécuter le code (typiquement **`Ackermann(5, 5)`**). Selon le langage employé, vous tomberez probablement sur des messages parlant d'une *pile d'appels* (*call stack* en anglais) trop remplie, ou d'un nombre trop important d'*appels* de fonctions, voire d'un problème de *stack frame*.
Lorsque l'on appelle une fonction ou une procédure, il est nécessaire pour l'ordinateur d'enregistrer le contexte dans lequel il était, et surtout, de préparer de l'espace pour l'exécution de la fonction/procédure (pour les variables, par exemple). Dans le cas d'une fonction récursive, à chaque appel récursif, il faut donc sauvegarder l'état des paramètres et variables, et rappeler la fonction avec de nouveaux paramètres et variables.
Lorsque ces appels successifs consomment trop de mémoire, le système d'exploitation peut refuser d'appeler des fonctions supplémentaires, et il déclenche donc un *dépassement de pile* (en anglais *stack overflow*).

Pour calculer la factorielle en récursif, le **`retourne`** du cas général contient une multiplication entre une valeur et un appel récursif. Il est donc nécessaire d'effectuer une multiplication avant de retourner la valeur finale. Afin de limiter la consommation mémoire, on peut effectuer de la *récursion terminale* en n'effectuant aucune opération dans le **`retourne`** qui effectue l'appel récursif.
La valeur finale n'est plus calculée au fur et à mesure des retours d'appels récursifs, mais directement lors de l'appel récursif le plus profond. Ainsi, chaque **`retourne`** ne sert qu'à renvoyer une valeur sans faire le moindre calcul.

```
algorithme fonction FactR : entier
  parametres locaux
    entier    n

debut
si (n == 0) OU (n == 1) alors
  retourne (1)
sinon
  retourne (n * FactR(n - 1))
fin si
fin algorithme fonction FactR
```

```
algorithme fonction FactRT : entier
  parametres locaux
    entier    n, acc

debut
si (n == 0) OU (n == 1) alors
  retourne (acc)
sinon
  retourne (FactRT(n - 1, n * acc))
fin si
fin algorithme fonction FactRT
```

Dans la version récursive terminale, on utilise un *accumulateur* qui stocke la valeur calculée. Ainsi, l'accumulateur est initialisée à une valeur neutre, et il est mis à jour au fur et à mesure des appels (et non pas des retours d'appels). Dès que le cas spécifique est rencontré, on renvoie directement cet accumulateur.

Étant donné que la fonction *factorielle* n'est censée prendre qu'un seul paramètre, il est nécessaire de fournir une fonction initialisant l'accumulateur et permettant de respecter le format attendu/ Pour cela, on écrit une *fonction chapeau*. Dans le cas de la factorielle, la valeur neutre à la multiplication est 1 (si on écrivait l'algorithme de la somme récursive, le neutre de l'addition est 0).
Celle-ci peut également servir à capturer les cas spécifiques et limiter les appels. Par exemple, dans les appels récursifs, on n'arrivera jamais au cas où N est égal à 0. Ainsi, on va immédiatement ce cas spécial dans la fonction chapeau, et limiter le nombre de tests dans la fonction récursive terminale.
On notera qu'un algorithme écrit en récursif terminal est facilement transformable en algorithme itératif : l'accumulateur devenant la variable évoluant au fur et à mesure des tours de boucle. La principale difficulté réside dans la transformation d'un algorithme récursif en récursif terminal.

```
algorithme fonction FChapo : entier
  parametres locaux
    entier    n

debut
si (n == 0) OU (n == 1) alors
  retourne (1)
sinon
  retourne (FactRT(n, 1))
fin si
fin algorithme fonction FChapo
```

```
algorithme fonction FactRT : entier
  parametres locaux
    entier    n, acc

debut
si (n == 1) alors
  retourne (acc)
sinon
  retourne (FactRT(n - 1, n * acc))
fin si
fin algorithme fonction FactRT
```

21) Réécrivez les algorithmes récursifs en récursifs terminaux. Attention, certains sont beaucoup plus complexes que d'autres. (N'essayez pas de convertir l'algorithme d'Ackermann en récursif terminal tant que vous débutez)

Ce document et ses illustrations ont été réalisés par Fabrice BOISSIER en octobre 2022.
La plupart des exercices sont inspirés du cahier d'algo de Nathalie "Junior" BOUQUET et Christophe "Krisboul" BOULLAY.
(dernière mise à jour octobre 2023)