[PROF] Logique et Récursivité
Fiche 2

Ce document a pour objectif de guider les enseignants pour le cours d'algorithmique. Il est déconseillé de le fournir aux étudiants, car il vise surtout à vous permettre de guider la séance. Vous n'êtes évidemment pas obligé de le suivre à la lettre (c'est même déconseillé, car cela va autant vous perturber vous que la classe : suivez votre chemin de pensée et/ou celui de la classe, et ensuite vérifiez que vous n'avez rien oublié).

La deuxième séance vise à découvrir la notion de récursivité, et de façon cachée à se rendre compte des effets de l'ordre des conditions à base de *if ... else*. Il ne faut pas hésiter à insister sur les conséquences des cascades de *if* imbriqués sur les cas absorbés dès le début/les cas non détectés par l'enchaînement sauvage de conditions.
Aucun ordinateur n'est nécessaire : vous pouvez les interdire pendant cette séance, néanmoins, n'hésitez pas à montrer ou à faire tester à l'un des étudiant l'algorithme de Ackermann avec les paramètres *(1, 1)*, *(2, 2)*, *(3, 3)*, puis *(4, 3)* en mesurant le temps passé (éventuellement avec la commande *time*). N'importe quelle implémentation sur OCaml ou Python suffit. N'hésitez pas à leur demander de tester le cas *(8, 8)* qui devrait faire déborder la pile d'appels.

1 Logique

- Explications de base : la *logique* est un domaine des mathématiques avec de nombreux sous-domaines et applications (logique de premier ordre, ...)
- (Instant Culture G : *La logique est initialement issue de la Philosophie dans l'antiquité afin de permettre d'établir des assertions et des raisonnements logiques/qui ne soient pas contradictoires... et sa formalisation l'a amenée à devenir une partie des mathématiques. Néanmoins il existe encore un champs de recherche « logique », propre à la Philosophie, enseigné dans les universités ainsi que des publications sous forme de livres dans des librairies spécialisées*)
- La logique est utilisée 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)
- Les *formules logiques* permettent de vérifier des assertions avec des opérateurs
- Opérateurs logiques et leurs tables de vérité : *NOT* (opérateur unaire), *AND*, *OR* (opérateurs binaires)
- Explication des *tables de vérité* : elles présentent tous les états possibles

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

- Rappel que *NOT*, *AND*, *OR* sont utilisés en algorithmique...
- ...mais il existe d'autres opérateurs logiques utilisés en mathématiques...
- ...ainsi qu'en électronique numérique, mais dans un format un peu spécial.
- Montrer les opérateurs *NAND*, *NOR*, et surtout *XOR*

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

— Montrez les symboles logiques :
- NON / Négation : $\neg$ autre notation : $\overline{A}$
- ET / Conjonction : $\wedge$ autre notation : $A \cdot B$ (agit comme un $\times$ entre les états)
- OU / Disjonction : $\vee$ autre notation : $A + B$ (agit comme un $+$ entre les états)
- Chaque notation vise un public/domaine précis :
  Mathématiques/logique ($\neg \wedge \vee$)
  Électronique numérique ($\overline{A} \cdot B$)
  Algorithmique (**`AND OR / && ||`**)

— Indiquez que distribuer un *NOT* sur un *AND*/*OR* inverse l'opérateur (théorème de De Morgan)
- $\neg(A \wedge B) = (\neg A) \vee (\neg B)$ (autre notation : $\overline{A \cdot B} = \overline{A} + \overline{B}$)
- $\neg(A \vee B) = (\neg A) \wedge (\neg B)$ (autre notation : $\overline{A + B} = \overline{A} \cdot \overline{B}$)

— Effectuez un exemple avec une table de vérité pour le prouver aux étudiants

— Expliquez les *conjonctions de cas* et les *disjonctions de cas**
- Conjonction de cas : $(A \vee B \vee C) \wedge (\neg A \vee B) \wedge ...$
- Disjonction de cas : $(A \wedge B \wedge C) \vee (\neg A \wedge B) \vee ...$

— Demandez à la classe lequel est le plus optimal et dans quel cas
- Conjonction de cas = une condition est fausse ⇒ tout est faux
- Disjonction de cas = une condition est vraie ⇒ tout est vrai

1.1 Exercices de logique

1.1.1 Exercices sur les conditions

1) Transformez les conditions suivantes en distribuant la négation :

```
int A, B, C, D;
```

a
```
if (! (A > 2) )
```

b
```
if (! ((A >= 2) AND (B < 3)) )
```

c
```
if (! ( ((A != B) OR (B == C)) AND (! ((A == C) OR (B != D))) ) )
```

2) Écrivez une condition à base de *AND*, *OR*, *NOT* reproduisant l'effet du *XOR* sur deux valeurs A et B

1.1.2 Algorithmes

3) Écrivez une fonction prenant 3 entiers différents en paramètre, et indiquant lequel est le plus petit/grand. *min(a, b, c)* *max(a, b, c)*

4) Écrivez une fonction prenant 3 entiers différents en paramètre, et renvoyant l'élément intermédiaire. *middle(a, b, c)*

5) 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), ...

6) Écrivez une fonction effectuant la somme des carrés des 2 plus grands/petits nombres parmi 3. *SumSquare(a, b, c)*

2 Récursivité

— Expliquer que la récursivité est simplement une fonction qui se rappelle elle-même
— Rappeler que certains l'ont vu en mathématiques avec le raisonnement par récurrence
— Demander quel était le principe du raisonnement par récurrence à la classe :
  • Très souvent le cas 0, ou le cas minimal/maximal
  • Le cas général entre l'itération N et l'itération N+1
— Écrire une fonction récursive se fait de la même manière :
  • Le(s) cas d'arrêt d'abord
  • Le cas général récursif en dernier
— *WARNING : Bien rappeler et expliquer qu'il est très important de mettre le cas d'arrêt d'abord !*
— Montrez ce qu'il se passerait si on mettait le cas d'arrêt *après* l'appel récursif général

2.1 Exercices récursifs

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

7) 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)

8) 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.

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

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

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

10) Écrivez l'algorithme récursif calculant la multiplication de deux entiers positifs, en n'utilisant que des additions et des soustractions.

11) 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.

12) 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.

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

13) 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 : $fibo(0) = fibo(1) = 1$)
$fibo(n) = fibo(n-1) + fibo(n-2)$

14) 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.]

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

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

16) 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 :

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

2.2 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.

17) 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)*

18) 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)*

19) 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)*

20) 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)*

21) 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.

22) 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 :

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

23) 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)*

3 Récursivité terminale

— Effectuez à la main au tableau la pile d'appels de **`Ackermann(5, 5)`**)... enfin quelques appels

— Expliquez le principe de la *pile d'appels* (*call stack* en anglais) et que celle-ci est limitée dans les ordinateurs

- Demandez à la classe pourquoi la pile d'appels est limitée

⇒ Parce que les ordinateurs ont une mémoire limitée

⇒ Parce que la mémoire contient d'autres données et informations, et que la pile est limitée pour ne jamais pouvoir écrire ailleurs

— Expliquer ce que contient la pile d'appel : les paramètres donnés à une fonction, la ligne de code où l'on en était, la valeur retournée par la fonction, ...

— Si la pile d'appels est trop remplie, à cause d'un excès d'appels de fonctions, alors cela déclenche un *dépassement de pile* (en anglais *stack overflow*)

— On peut réduire l'usage de la pile d'appels lorsque l'on effectue de la récursivité en utilisant une technique particulière : la *récursivité terminale*

— Montrez que Fibonacci en récursif classique se bloque sur une multiplication entre une valeur et un appel récursif

— Expliquer que l'ordinateur doit donc attendre le retour de la fonction *avant* de pouvoir faire sa multiplication

— Entre chaque retour de fonction, il y a donc des opérations effectuées...

— L'écriture de fonctions récursives terminales va éviter cela, et permettre de ne faire que renvoyer des valeurs en cascade

— Plus précisément, on va construire la réponse finale au fur et à mesure, et dans l'appel le plus profond on aura le résultat final

```
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
```

— Explicitez le rôle de l'*accumulateur* et le fait qu'il est renvoyé

— Les conditons vont pouvoir capter plus de cas et renvoyer l'accumulateur plus tôt (pas besoin d'additionner 0 ou de multiplier par 1 un résultat avant de le renvoyer)

— Rappelez que lorsque les consignes d'exercices, d'examens, ou d'API du monde réel, donnent un prototype précis de fonction à respecter : il faut le respecter

— Demander à la classe comment ajouter un accumulateur tout en respectant le prototype strict qui n'en inclut pas

⇒ avec une fonction chapeau qui va préparer l'accumulateur

```
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
```

— Explicitez le fait que la valeur initiale de l'accumulateur va dépendre de la logique employée dans l'algorithme récursif

— Expliquez qu'un algorithme récursif terminal se transforme facilement en algorithme itératif
  - L'accumulateur devenant une variable évoluant au fur et à mesure des tours de boucle

— La seule difficulté est de passer d'un algorithme récursif à un algorithme récursif terminal

24) 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 2024.
La plupart des exercices sont inspirés du cahier d'algo de Nathalie "Junior" BOUQUET et Christophe "Krisboul" BOULLAY.
(dernière mise à jour octobre 2024)