AGRO CAMPUS OUEST

Algorithmique avec R

T. Guyet

Version v1, Dernière modification : 4 mars 2014

Table des matières

Chapitre I

Introduction

1 Objectifs du cours

Le cours ne concerne que l'algorithmique impérative. D'autres paradigmes de programmations existent (langages objet, langages déclaratifs, programmation événementiel), mais nous ne nous intéresserons qu'à l'algorithmique impérative. On parle de langage impératif (ou algorithme impératif) parce que le programme contient des ordres que l'exécutant de l'algorithme devra suivre à la lettre.
Les objectifs du cours sont les suivants :
- connaître les éléments de base de l'algorithmique (principes et écriture en pseudo-code et en R) : variables (tableau et data.frames), structures de contrôle (boucles, alternatives), conditions, fonctions.
- connaître quelques algorithmes élémentaires de manipulation de tableau R
  - algorithmes de recherche d'un élément sans ou avec mémoire (avec : min/max)
- comprendre un programme écris en R, incluant des fonctions
  - comprendre l'utilisation des fonctions de vectorisation
- savoir écrire et exécuter un programme en R dans l'environnement R ou RStudio, incluant
  - utilisation de l'indentation,
  - écriture de commentaires dans le code et les fonctions
- savoir adapter et composer des algorithmes (écrits en R ou en pseudo-code) pour répondre à un besoin

Les difficultés intrinsèques à l'algorithmique sont :
- La machine a des capacités limités : il faut arriver à recomposer une fonctionnalité complexe à partir d'opération souvent élémentaires
- L'algorithme est conçu en différé : il y a toujours deux étapes dans la programmation 1) la création du programme et 2) son exécution. Tout le travail du programmeur est d'anticiper de manière exhaustive les situations que va rencontrer la machine.

Les difficultés intrinsèques à la programmation sont :
- L'ordinateur est une machine formelle : il faut se plier à ces formalismes,
- La programmation est un langage, il faut en connaître son vocabulaire et sa grammaire pour pouvoir comprendre et discuter.

2 Présentation du polycopié

Ce polycopié est une introduction à l'algorithmique. Il est organisé comme une sorte de *cook-book* d'algorithmes. En même temps qu'on vous introduit les concepts élémentaires de l'algorithmique, il illustre ces concepts d'algorithmique par des exemples qui doivent servir de recettes de base pour concevoir des algorithmes plus complexes. Pour chacun des exemples, on explique plus ou moins précisément comme ils ont été construits pour vous initier la démarche complexe de la création d'un algorithme. Avant tout, il faut chercher à les comprendre pour être capable de les reproduire, de les adapter à vos besoins, à vos données. Il faut que vous les ayez suffisamment intégrés pour que lorsque vous rencontrerez un nouveau problème, vous puissiez vous dire qu'il s'agit du même problème que l'algorithme *machin* ... mais avec un petit truc qui change. Il est certain que vous trouverez qu'il y a beaucoup d'algorithmes présentés, mais ce ne sont que les algorithmes élémentaires ... après ce ne sera qu'une question de pratique pour les maîtriser tous et étendre vos connaissances avec de nouveaux algorithmes utiles.

La première partie présente l'algorithmique de base. On reviendra assez longuement sur l'opération d'affectation. Ensuite, on passe en revue les différentes structures de contrôle qui se rencontrent classiquement en algorithmique (système de Hoarce). Deux outils sont utilisé lors de la présentation de cette première partie : les tableaux d'évolution de variables, les commentaires d'invariants (je commence ces commentaires par un *#ICI*). Pour le premier, il est très utile de pouvoir l'utiliser pour comprendre le fonctionnement d'un algorithme sur des exemples. Le second est un outil utilisé par les programmeurs qui s'intéressent aux algorithmes pour assurer le bon fonctionnement de leurs programmes. Ils permettent de raisonner sur les programmes.
La seconde partie présente des éléments d'algorithmiques sur les tableaux. La plupart des méthodes sont présentées sur des tableaux unidimensionnels. Les tableaux permettent de faire des traitements de masse ... soit exactement ce qu'on demande à une ordinateur. Il s'agit ici de connaître les traitements élémentaires qui peuvent

être réalisés sur des tableaux pour automatiser des traitements.
La troisième partie présente le principe de la décomposition d'un code et l'utilisation de fonctions.

Ce polycopié utilise la syntaxe de R pour présente l'algorithmique impérative. Pour les syntaxes spécifiques à R, il est conseillé d'avoir pris connaissance du polycopié de R avant, et d'y revenir au besoin.

Chapitre

II Algorithmique de base

Un algorithme ou un programme est la description d'une procédure mécanique. Cette description abstraite doit permettre à une machine de déterminer de manière complète et non-ambiguë quelles sont les opérations élémentaires que la machine doit faire (impérativement). Dans le cas de l'algorithmique informatique, la machine est un ordinateur.
L'exécution d'un algorithme réalise mécaniquement la suite d'opération définie dans l'algorithme. Lorsque le programme a exécuté toutes les tâches qu'il avait à faire, on dit qu'il "termine". Dans l'ensemble des algorithmes que vous pouvez écrire, seule une petite partie termine effectivement.
Un algorithme implémente une "fonctionnalité" : lorsque le programme termine, il a, au final, réalisé une tâche. La fonctionnalité de l'algorithme, c'est un peu la description de la tâche qui va être accomplie : connaissant les données d'entrée, quelles vont être les sorties ? Tout l'intérêt de la programmation est de construire des algorithmes qui vont faire faire la tâche souhaitée à l'ordinateur. On verra un peu plus loin qu'il est alors intéressant de pouvoir prouver que le programme qu'on a écris réalise bien la tâche voulue. On parle alors de **preuve de programme**.

1 Variables

Un programme informatique est une automatisation de traitements sur des variables. La principale activité de l'ordinateur consiste à recopier des variables, mettre des valeurs dans les variables, faire des opérations (arithmétiques) entre les variables. C'est donc un élément central dans les langages de programmation.
Une variable est composé de trois éléments :
– un nom de variable : c'est ce nom qu'on utilise dans un algorithme
– une valeur : c'est ce que "contient" la variable.
– un type : c'est la "nature" de la valeur contenue dans la variable. Ce peut être par exemple un nombre entier, ou bien une lettre ou encore un vecteur de nombres à virgules, ...

La notion de typage de la variable peut être comparé à celui des notations mathématiques. Lorsqu'on écrit "soit $x \in \mathbb{R}$", on écrit que x est un variable de type 'nombre réel', mais on ne lui a pas donné de valeur. Les variables informatique ont toujours un valeur quite à définir un valeur particulière signifiant qu'il n'y a pas de valeur ! En R, la valeur `NA` a un peu cet office.

On peut donner également quelques informations sur les noms des variables. En programmation, comme en algorithmique, il est intéressant de définir des "convention de nommage", c'est-à-dire des règles qu'on s'impose pour rendre les variables plus facilement compréhensible. Voici quelques exemples de convention :
– éviter l'usage de noms de variables qui ne permettent pas de savoir ce que contient la variables
– conserver l'usage des `i` et `j` pour des boucles très locales
– commencer toutes les noms de variable par une lettre minuscule, et les autres débuts de mots en majuscules, par exemple `maVariable`
– commencer le nom d'une variable par une lettre qui précise sont type, par exemple `dMaVariable` le `d` indiquant qu'il s'agit d'un entier.
– etc.
L'objectif des conventions de nommage n'est pas d'embêter le programmeur, c'est de faciliter la vie à celui qui va lire le programme. Le premier étant fréquemment le second (ou un proche professionnel du second), il est plus qu'intéressant de les utiliser. Ces conventions sont particulièrement intéressantes dans le cas du langage R qui est plutôt laxiste sur le typage.

1.1 Opération sur les variables

1.1.1 Affectation

L'opération principale qui peut être faire sur une variable est l'**affectation**, c'est-à-dire l'attribution d'une valeur à une variable. En R, on note <− l'opération d'une affectation [1]. Il doit nécessairement avoir une variable à gauche de la flèche. Le sens de la flèche indique le sens de l'affectation : c'est la variable située à de gauche qui se voit attribuer une nouvelle valeur : celle se trouvant à droite.
On peut distingueur trois sortes d'affectation :
– l'affectation par une valeur,
– l'affectation par une autre variable (ou recopie de variable),
– l'affectation par le résultat d'une opération.

Exemple 1 - Affectation d'une variable

L'exemple ci-dessous illustre successivement les trois types d'affectation évoqué ci-dessus.

```
x <- 3
y <- x
z <- x+y
```

Dans le premier cas, `x` *va se voir attribuer la valeur* 3*. Dans le second cas,* `y` *va se voir attribuer la valeur contenue dans la variable b, c'est à dire également la valeur* 3*. Dans le troisième cas,* `z` *va être affecté par le résultat de l'opération valeur* 3*.*

! Attention ! - Une valeur n'a qu'un seul contenu

Lors de l'affectation d'une variable, la valeur affectée remplace la valeur que contenait précédemment la variable. En supposant que les deux opérations suivantes soient successives :

```
a <- 3
# ICI a contient la valeur 3
a <- 5
# ICI a contient la valeur 5
```

En fait, il n'est pas nécessaire de faire cette distinction un peu artificielle des trois types d'affectation, il est plus pertinent de comprendre que l'affectation se passe en deux temps :

1. la partie à droite de la flèche est évaluée : la machine calcule une valeur à partir de la partie droite, quelque soit sa complexité !
2. la partie à gauche est ensuite affectée par la valeur calculée dans la partie droite.

Un cas particulier très courant est l'auto-affectation : la valeur affectée à une variable dépend de la valeur de cette même variable. Dans la mesure où il y a bien deux étapes dans l'affectation, cela ne pose pas de problème : la partie droite est évaluée avec la valeur originale de la variable (avant modification), et une fois l'opération terminée, alors

Exemple 2 - Auto-affectation

L'exemple ci-dessous illustre deux auto-affectation. À la première ligne, `a` *vaut 3. À la seconde ligne, la valeur 3 est utilisée pour* `a` *dans le calcul de* $2 * a + 3$ *et ensuite, le résultat (*9*) est affecté à* `a`*. De même dans le dernier cas, où la valeur de* `a` *est utilisée pour chaque occurrence de* `a` *dans la partie droite, puis le résultat du calcul donne la nouvelle valeur de* `a`*.*

```
a <- 3
a <- 2*a+3
a <- 10*a-4**a
```

1.1.2 Opération entre variables

Reste maintenant à décrire tout ce qui peut être fait comme manipulation entre les variables. Pour cela, il faut se référer au langage étudié.

1. Dans beaucoup de langage, l'opérateur = est utilisé. <− comme = peuvent être utilisé en R.

Dans la suite de ce cours d'algorithmique, on supposera qu'il est possible de faire les opérations arithmétqiues usuelles (sauf si indication contraire d'un exercice) : addition, multiplication, ... La notation `**` peut être utilisée pour désigner le calcul d'une exponentiation.
L'usage des parenthèses est de mise pour gérer les priorités usuelles des opérations.

Lorsque nécessaire, vous pourrez utiliser les fonctions mathématiques usuelles (`cos`, `sqrt` pour la racine carrée, ...).

Pour les types de données autres que les nombres, se référer au langage.

1.2 Portée de variable

Cette section introduit des notions qui seront peu utilisées en langage R. Vous pouvez passer cette section et y revenir lorsqu'elle se montrera plus pertinente.
La portée d'une variable désigne l'étendue du code dans lequel une variable qui a été définie à un endroit donné peut être utilisée. On différencie généralement deux types de portée :
– la variable locale : elle ne pourra être utilisée uniquement dans une partie du code, qu'on pourrait définir comme un "contexte". Dans ce contexte, comme pour toute variable, elle ne peut être utilisée qu'à partir du moment où elle a été définie (affectation initiale), mais en dehors du contexte, la variable n'existe plus.
– la variable globale : à partir de l'endroit où elle a été définie dans le code, elle pourra être utilisée dans toute la suite de l'exécution du code. Par exemple, lorsque vous définissez une variable dans la console R elle peut être utilisée dans toutes les commandes ultérieures. On peut se représenter les choses en imaginant qu'il existe un "contexte" général qui est toujours accessible.

Remarque 2 - Masquage d'une variable

La notion de portée d'une variable est également associée à la notion de masquage. Si une variable a été définie dans un contexte, elle peut être redéfinie dans un sous-contexte. Dans ce cas, la variable du sous-contexte masque temporairement la variable du contexte plus général.

Il est possible d'utiliser ces modes de fonctionnement pour écrire des algorithmes plus lisibles ou plus efficaces. Leur usage n'est pas nécessairement facile de prime abord. Mais il est plus souvent intéressant de connaître ces mécanismes pour comprendre certains comportement des programmes et éviter des erreurs.
Dans la plupart des langages, les problèmes de portée de variables interviennent très souvent. Dans le cas du langage R, ils n'interviennent que lors de l'utilisation de fonction. Un "contexte", tel que décris précédemment est un corps de fonction. Les deux niveaux de contexte qui existent sont uniquement le contexte de la fonction et le contexte général. Nous reviendrons donc sur ces notions lors du chapitre 1 sur les fonctions.

2 Structures de contrôle

Les structures de contrôle sont les éléments clés de l'algorithmiques. Ce sont des éléments du programme qui vont "contrôler" l'enchaînement des opérations élémentaires que va réaliser l'ordinateur.
Les structures de contrôle que nous allons voir sont :
– la séquence,
– l'alternative,
– l'itération,
– le branchement
Ces quatre structures de contrôle forment le **système de Hoare**. Ce système est à la fois très générique et formellement très intéressant pour raisonner sur les programmes.

2.1 Instruction

Une instruction est une opération élémentaire que sait faire une machine et qui fait quelque chose. Dans notre cas, nous avons vu pour le moment une seul instruction : l'affectation. Les opérations sur les variables n'auront aucune conséquence si le résultat de ces opérations n'est pas enregistrer dans une variable. La seule vraie instruction est donc bien l'affectation.
Par la suite, on verra deux autres instructions :
– la lecture d'une valeur au clavier, dans un fichier ou une base de données
– l'affichage d'une valeur à l'écrans
Nous ferons toute l'algorithmique avec uniquement ces instructions.

2.2 La séquence et le bloc d'instructions

2.2.1 Séquence

La séquence est la structure de contrôle la plus simple qui indique que les instructions sont faites successivement. La séquence d'exécution des instructions est exprimée dans un programme par la succession des lignes du programme.
La contrainte imposée par la séquences est qu'une instruction ne peut être exécuté tant que l'instruction précédente n'a pas été finalisée.

Exemple 3 - Séquence

Dans l'exemple ci-dessous, on illustre un programme contenant une séquence de trois instructions. Le ***tableau de variables*** *illustre l'exécution du programme. Les opérations sont effectuées les unes après les autres. Ainsi, l'opération d'addition ne se fait qu'à la fin, après l'instruction d'affectation de la valeur 5 à* **a**.
Vous noterez également que lorsque **a** *change de valeur alors* **b** *n'est pas modifié ! Rien ne lie la valeur contenue dans* **b** *et* **a**
On notera qu'initialement on ne sait pas ce que contiennent les variables, dans le tableau, on indique alors un point d'intérrogation.

```
1  a <- 3
2  b <- a
3  a <- 5
4  d <- a+b
```

line	*a*	*b*	*d*
1	*3*	*?*	*?*
2	*3*	*3*	*?*
3	*5*	*3*	*?*
3	*5*	*3*	*8*

2.3 L'alternative

L'alternative consiste à proposer un comportement differentié du programme en fonction des conditions d'exécution de celui-ci.

2.3.1 La structure de contrôle if

La notation consacrée est la suivantes :

```
if ( condition ) {
   # bloc d' instructions  realise  si  la  condition  est  vraie
   ...
} else {
   # bloc d' instructions  realise  si  la  condition  est  fausse
   ...
}
```

- la condition est un test : c'est une valeur, une variable ou une opération qui sera évalué à vrai (**T**) ou faux (**F**).
- les deux blocs d'instructions correspondent à ce que doit exécuté la machine dans le cas ou la condition est vraie, ou dans le cas où la condition est fausse.

Exemple 4 - Alternative

Dans l'exemple ci-dessous, on génére aléatoirement un nombre entre 0 et 1, puis en fonction de la valeur généré, il attribut une valeur à **p** *de manière différente.*
Dans les tableaux de variables, on illustre deux exécutions différentes. Dans les tableaux de variable, on affiche des colonnes spécifiques pour les tests du programme. Ces colonnes ne peuvent prendre que les valeurs T ou F. La valeur initiale de y est générée aléatoirement, on signale cela par un point d'exclamation lors de l'initialisation.

```
1  #Generation d'un nombre aleatoire  entre  0  et  1
2  y <- runif(1)
3
4  if ( y<0.5 ) {
5    # ICI y<0.5
6    p <- y
7  } else {
8    # ICI y>=0.5 et y<1
9    p <- 1-y
10 }
11 #ICI p est  entre  0  et  0.5
12
13 #Affichage du  resultat
14 print (p)
```

Exécution 1 :

line	*y*	*p*	*y<0.5*
2	*0.08(!)*	*?*	*-*
4	*0.08*	*?*	*T*
5-6	*0.08*	*0.08*	*-*
11-14	*0.08*	*0.08*	*-*

Exécution 2 :

line	*y*	*p*	*y<0.5*
2	*0.77(!)*	*?*	*-*
4	*0.77*	*?*	*F*
8-9	*0.77*	*0.33*	*-*
11-14	*0.77*	*0.33*	*-*

En conclusion de cet exemple, on peut constater que "ce programme génére une variable **p** *dont la distribution est uniforme entre 0 et 0.5, et on l'affiche". À l'aide des commentaires, on voit assez rapidement que* **p** *est entre 0 et 0.5, il faudrait raisonner un peu plus pour avoir l'uniformité ... mais c'est sommes toutes assez simple de s'en apercevoir.*

2.3.2 Le else n'est pas indispensable

On peut très bien écrire des if sans else. Ceci est utile lorsqu'un comportement du programme est nécessaire dans un cas, mais que le cas contraire demande de ne rien faire ! Dans ce cas, on n'écrit pas :

```
if( condition ) {
   #bloc d' instruction
   ...
} else {
   # ne rien faire ~!!
}
```

Exemple 5 - Alternative sans else

L'exemple ci-dessous à la même fonctionnalité que l'algorithme de l'exemple 2.3.1 sans utiliser deux cas distincts. En fait, le principe de l'algorithme ci-dessous est de définir un cas, par défaut, et si on en sort, alors on fait une rectification. Ici, on veut que **p** *soit une valeur entre 0 et 0.5 à partir du générateur aléatoire entre 0 et 1. Si la valeur de* **p** *est inférieure à 0.5, on n'a rien de plus à faire, mais si ce n'est pas le cas, on modifie la valeur de* **p** *pour la ramener entre 0 et 0.5.*

```
#Generation d'un nombre aleatoire entre 0 et 1
p <- runif(1)

if ( p>=0.5 ) {
  # ICI p>=0.5
  p <- 1-p
  # ICI p<=0.5
}
#ICI p est entre 0 et 0.5

#Affichage du resultat
print (p)
```

2.4 Alternative entre plus de 2 cas distincts

Admettons que vous souhaitiez distinguer 4 cas que peut prendre un entier x :
– $x \in]\infty, -5]$: traitement 1
– $x \in]-5, 0]$: traitement 2
– $x \in]0, 45]$: traitement 3
– $x \in]45, +\infty]$: traitement 4
Notez que j'ai pris soin de définir mes cas de manière **non-ambigüe** (pour une valeur de x il y a au plus 1 cas à traiter) et **exhaustive** (pour chaque valeur de x il y a au moins 1 cas à traiter). Ces deux contraintes sont impératives pour assurer que votre programme est bien complet et ne risque pas de provoquer des erreurs ou pire, des comportements inattendues.
Jusque là, nous n'avions rencontré que des dichotomies, c'est-à-dire des tests de la forme $a < 34$ qui faisait la séparation de deux cas : la non-ambigüité et la complétude de traitement étaient alors assurés !

Une première solution pour sélectionner l'alternative est de transformer ces 4 cas en dichotomies : d'abord séparer entre les valeurs >0 et <0, puis séparer de nouveaux chacun des deux cas en deux sous-cas. Cette solution revient à faire de l'imbrication de conditions comme montré ci-dessous. Vous noterez que ce programme est correct (il fait ce qui était demandé) dans la mesure où ou a bien pris soin des limites dans les tests.

```
if( x<0 ) {
  # ICI il faut le comportement 1 ou 2
  if( x<= -5 ) {
    print("traitement 1")
  } else {
    print("traitement 2")
  }
} else {
```

```
  # ICI il faut le comportement 3 ou 4
  if ( x>45 ) {
    print("traitement 4")
  } else {
    print("traitement 3")
  }
}
```

Une autre solution sera d'utiliser le caractère "ordonné" des réels (cf. programme ci-dessous). Dans les exemples ci-dessous, les blocs d'instructions sont exclusives (propriété du `else if`). Si on prend la ligne 6, on n'accède à cette ligne 6 que si deux conditions sont respectées :
– les tests précédents du `else if` étaient tous faux, dont $x > -5$
– si le test de la ligne 4 est vrai : $x <= 0$
on retrouve donc bien les propriétés données dans les commentaires.
Dans le tableau d'évolution de variables avec $x = -3$, on commence par tester $x <= -5$, comme le test est faux, on passe au test suivant $x <= 0$ qui est vrai. On rentre alors dans le bloc d'instructions qui réalise le traitement 2, puis une fois ce traitement effectué, on ne teste même pas la ligne 7, mais on va directement à la fin de la structure de contrôle pour exécuter la suite du programme.

```
1  if ( x<= -5 ) {
2    # ICI  -x <= -5
3    print("traitement 1")
4  } else if ( x<= 0 ) {
5    # ICI  -5 < x <=0
6    print("traitement 2")
7  } else if ( x<= 45 ) {
8    # ICI  0 < x <= 45
9    print("traitement 3")
10 } else {
11   # ICI  45 < x
12   print("traitement 4")
13 }
```

line	x	x<= -5	x<= 0	x<= 45
1	-3	F	-	-
4	-3	-	T	-
5-6	-3	-	-	-
13	-3	-	-	-

Lequel des deux programmes est le meilleur ? ?
D'un point de vue de la lecture du programme, le second est nettement plus lisible. On comprend de suite qu'il y a 4 cas séparés. L'imbrication du premier programme n'est pas nécessairement facile à lire.
Si on s'intéresse aux temps de calcul, ce qui est intéressant de regarder, se sont les performances en moyenne. On comprend bien que quelque soit la valeur de x, le nombre de comparaison à faire pour le premier programme est de 2. En revanche, dans le second programme, il faudra faire simplement 1 comparaison si $x <= -5$ mais 3 comparaisons si $x > 45$. Si les x sont très souvent au dessus de 45, mais jamais en dessous de -5, il faudra peut être préférer la première approche. Mais sinon, il n'y a pas de résultat immédiat.

N'hésitez pas à faire appel à un cas par défaut ! ! Dans le cas du second programme, il est inutile d'ajouter une condition pour le dernier cas, si vous avez prévu une partition des valeurs qui peuvent être proposées, alors il est possible d'utiliser cette propriété de partition du domaine pour éviter un dernier test.
Mais dans certains cas, tous les cas ne peuvent pas être traités exhaustivement. C'est le cas, par exemple, pour des variables qui contiennent du texte ! Le seul test à faire est l'égalité et dans ce cas, vous ne pouvez pas anticiper toutes les valeurs possibles !
Tout ce qui n'a pas à être traité doit se retrouvé dans un cas "poubelle", le cas par défaut.

Exemple 6 - Tester des textes : espace de valeurs infini, non ordonné

Supposons que dans le cas où une variable **Texte** *de type texte contienne la valeur "toto", alors il faut faire le traitement 1 et si la valeur est "tutu" alors il faut faire le traitement 2.*
Aller hop, on programme cela rapidement :

```
if ( Texte== "toto" ) {
  print ("traitement 1")
}

if ( Texte== "tutu" ) {
  print ("traitement 2")
}
```

Mais peut-on se contenter de cela ? NON ! Que va-t-il se passer si **Texte** *vaut "foo" ou "bar" ? ? ? Dans le cas ci-dessous, l'utilisation de la structure de contrôle avec* `else if` *a deux avantages :*
– *on utilise la propriété que "toto" est différent de "tutu" pour ne pas avoir à faire le second test d'égalité si le premier était vrai !*

– *on utilise le* else *pour faire un implémenter un comportement par défaut. Ici, il s'agit d'afficher un message d'erreur.*

```
if ( Texte== "toto" ) {
  print ("traitement 1")
} else  if ( Texte== "tutu" ) {
  print ("traitement 2")
} else  {
  print ("erreur : la valeur est inattendue~!")
}
```

Remarque 3 - Quelques mots sur le switch

Dans des cas comme celui de l'exemple 2.4 où seule des égalités sont utilisées, il est possible d'utiliser l'instruction switch.

2.4.1 Combinaison de conditions sur plusieurs variables

Les espaces à séparer ne sont pas toujours unidimensionnels, c'est à dire que les traitements à faire ne dépendent pas nécessairement d'une seule variable, mais de plusieurs variables. Dans ce cas, il faut écrire des conditions qui dépendent de plusieurs variables.
Deux solutions s'offrent à vous pour écrire de tels structures de programme :
– faire des tests qui combinent plusieurs critères,
– faire des imbrications de if avec des tests plus simples.
Le premier cas est très générique. Si je pense qu'il est possible de se passer de la première solution à la seconde de manière équivalente. Le second cas est à conserver, de préférence, aux cas de variable indépendante vis-à-vis de la séparation des cas.

Exemple 7 - Séparation avec variables indépendantes

Commençons par nous intéresser à des variables indépendantes vis-à-vis de la séparation des cas.

```
1  #Generation d'un nombre aleatoire entre 0 et 1
2  y <- runif(1)
3
4  if ( y<0.5 ) {
5    # ICI y<0.5 quelque soit la valeur de y
6    # La proba d'avoir ce cas est donc 0.5
7    p <- 2
8  } else  {
9    x <- runif(1)
10   if ( x<0.7 ) {
11     # ICI y>=0.5 et x<0.7
12     # La proba d'avoir ce cas est donc 0.5*0.7=0.35
13     p <- -x
14   } else  {
15     # ICI y>=0.5 et x>=0.7
16     # La proba d'avoir ce cas est donc 0.5*0.3=0.15
17     p <- 2*x
18   }
19 }
20
21 #Affichage du resultat
22 print (p)
```

Exécution 1 :

line	*x*	*y*	*p*	*y<0.5*	*x<0.7*
1	*?*	*0.08(!)*	*?*	-	-
2	*?*	*0.08*	*?*	-	-
4	*?*	*0.08*	*?*	*T*	-
7	*?*	*0.08*	*2*	-	-
22	*?*	*0.08*	*2*	-	-

Exécution 2 :

line	*x*	*y*	*p*	*y<0.5*	*x<0.7*
1	*?*	*0.77(!)*	*?*	-	-
2	*?*	*0.77*	*?*	-	-
4	*?*	*0.77*	*?*	*T*	-
9	*0.83(!)*	*0.77*	*?*	-	-
10	*0.83*	*0.77*	*?*	-	*F*
16	*0.83*	*0.77*	*1.66*	-	-
22	*0.83*	*0.77*	*1.66*	-	-

On peut transforme le programme ci-dessous en un programme comme ci-dessous. Vérifions que le cas du else *correspond bien que au cas attendu pour attribuer la valeur 2 à* p*. Si on traite la ligne 11, alors c'est que* $y >= 0.5 \wedge x < 0.7$ *est faux et que* $y >= 0.5 \wedge x >= 0.7$ *est faux. On a donc* $y < 0.5 \vee x >= 0.7 \wedge y < 0.5 \vee x < 0.7$*, par distributivité, on a* $y < 0.5 \wedge (x >= 0.7 \vee x < 0.7)$ *et donc en simplifiant* $y < 0.5$*. Le commentaire de la ligne 10 est donc correct.*

```
1  y <- runif(1)
2  x <- runif(1)
3
4  if ( y>=0.5 && x<0.7 ) {
5      # ICI y>=0.5 et x<0.7
```

```
    p <- -x
} else if ( y>=0.5 && x>=0.7 ) {
    # ICI y>=0.5 et x>=0.7
    p <- 2*x
} else {
  # ICI y<0.5 quelque soit la valeur de y
  p <- 2
}

print(p)
```

Même question que précédemment : lequel est le mieux ? Personnellement, j'affiche ma préférence esthétique sur le second code à la condition qu'il soit commenté !! Sinon, les performances dépendent de beaucoup de chose ... elles seront au final très comparables ...

Pour le cas de variables liées, il est souvent nécessaire de faire appel à des conditions qui font intervenir des expressions arithmétiques qui combinent plusieurs variables. En fait, par rapport à l'exemple précédent, c'est assez différent, puisque la combinaison des variables dans une condition ne pose pas les mêmes problèmes de la combinaison de plusieurs conditions.

Exemple 8

Exemple de condition "complexe" faisant intervenir une condition qui combine linéairement deux variables.

```
y <- runif(1)
x <- runif(1)

if ( 2*y+3*x>=0.5 ) {
  p <- 1
} else {
  p <- 2
}
print(p)
```

2.5 L'iteration

La structure de contrôle d'iteration permet d'écrire dans un programme qu'un bloc d'instructions doit être répété plusieurs fois.

2.5.1 Les boucles while

Avec les boucles while le bloc d'instructions est repétée **tant qu'une condition est vraie**.
Dans l'exemple ci-dessous, on souhaite faire un programme qui calcule le premier nombre supérieur à 20 de la forme $3 * 2^n$ avec $n \in \mathbb{N}$. On procède par itération en testant successivement toutes les valeurs de n de manière croissante. Comme $3 * 2^n$ est une fonction strictement croissante avec n, on est sûr que si $3 * 2^n \geq 20$ alors il n'y aura pas aucune solution pour n supérieurs. Ceci nous permet de définir un critère d'arrêt tout en étant sûr de la complétude de l'algorithme.

```
a <- 3
n <- 1
while( a<20 ) {
  # ICI n indique le nombre de passage dans la boucle
  # ICI a vaut 3*2^(n-1)
  a <- a*2
  n <- n+1
}
# ICI n indique le nombre de passage dans la boucle + 1
print(n)
print(a)
```

line	a	n	a<20
1-2	3	1	-
3	3	1	T
5	6	1	-
6	6	2	-
3	6	2	T
5	12	2	-
6	12	3	-
3	12	3	T
5	24	3	-
6	24	4	-
3	24	4	F
7-9	24	4	-

L'exemple de boucle utilise `n` comme **indice de boucle**. C'est une variable qui sert à savoir, à chaque tour de boucle, où en est l'exécution de celle-ci. Ici, l'indice est un compteur de boucle, c'est le cas le plus général.

L'utilisation d'un indice doit rapidement devenir naturel. C'est un outil indispensable pour l'utilisation des boucles. Il faut noter que l'utilisation d'un indice de boucle implique trois éléments importants :
– l'initialisation de la variable (cf. ligne 2)
– l'évolution de la variable à chaque tour de boucle (cf. ligne 7). Ici, il s'agit d'une **incrémentation** simple (+1), on verra au fur et à mesure des exemples que l'évolution de la variable peut être plus complexe.
– l'utilisation de la variable dans la condition d'arrêt (pas utilisé dans notre exemple)
La variable `a` correspond également à une sorte d'indice de boucle dont l'évolution à chaque tour n'est pas linéaire.

Remarque 4 - Invariants de boucle

Vous noterez que deux lignes de commentaires ont été insérées dans la boucle. Ces commentaires expriment des invariants de boucle, c'est-à-dire des assertions [2] qui sont vraie à chaque tour de boucle. Les assertions intéressantes sont bien évidemment celles qui sont informatives par rapport au déroulement du programme soit parce qu'elles permettent de déterminer les arrêts de la boucle localement, soit qu'elles permettent de déterminer la fonctionnalité globale du programme.
La place de l'assertion à son importance. Généralement, elle est placée en début de boucle, mais on pourrait avoir des assertions en début et fin de boucle.
Dans l'exemple de boucle, l'assertion intéressante est celle qui affirme que $a = 3 * 2^(n-1)$.
La vérification d'un invariant de boucle se fait par un raisonnement par récurrence.

En algorithmique, lorsqu'on conçoit une itération, il y a toujours trois cas à considérer de manière spécifique :

1. le début de la boucle : c'est-à-dire ce qui se passe lorsqu'on fait pour la première fois le test ou qu'on rentre pour la première fois dans la boucle. C'est souvent un cas particulier de traitement du cas général.
2. le cas général de l'itération : c'est le cas de l'évaluation de la condition et de l'exécution du bloc d'instructions le plus courant
3. la fin de la boucle : c'est-à-dire ce qui se passe lors qu'on doit arriver à la fin du traitement : il n'est pas rare que les conditions d'arrêt soient mals définies et que le traitement "déborde" ou qu'il ne soit pas complet.

Une attention particulière doit être apporté au traitement des bords (début et fin) pour assurer que votre programme réalise bien la tâche demandée. On retrouve ici les spécificités d'un raisonnement par récurrence où un cas premier doit être traité avant de traiter le cas de récurrence.

Exemple 9 - Attention aux bords ...

Les boucles infinies Lors de la preuve de terminaison d'un programme, il faut avoir une attention particulière à la terminaison d'une boucle. Il faut toujours bien faire attention à ce que le test d'arrêt puisse est vrai.

Exemple 10 - Boucles infinies

Un premier exemple de boucle infinie triviale ... La condition `T` *n'est trivialement jamais fausse, donc ça ne s'arrête jamais. On verra plus tard que l'utilisation de cette boucle sans condition pourra être utile. Il est même possible d'écrire cette même boucle avec un* `repeat`*.*

```
while( T ) {
  cat("boucle\n");
}
```

```
repeat {
  cat("boucle\n");
}
```

Un second exemple de boucle infinie, moins évident, mais qui est une erreur classique où le programmeur utilise une condition d'arrêt un peu trop strict. Une condition du `while` *sur un nombre de tour de boucle peut être écrite* `i<n` *où* `i` *est l'indice de boucle. Certains programmeurs utilisent la condition suivante* `i!=n` *pour indiquer que le programme boucle tant que* `i` *est différent de* `n`*. Dans la plupart des cas, ça fonctionne, mais cette mauvaise habitude peu conduire à écrire le programme ci-dessous.*

```
1 i<-0
2 n<-45
3 while( i!=n ) {
4   cat("boucle\n");
5   i<-i+2
6 }
```

2. Une assertion est une affirmation qui est vérifiée (*c.-à-d.* vraie).

```
<-scan
```

Remarque 5 - Arrêter l'exécution d'une boucle infinie

Lors de l'exécution d'un script R qui contient une boucle infine, votre programme aura soit tendance à ne pas vouloir s'arrêter, soit produire des affichages à n'en plus finir. Il faut alors forcer l'arrêt de l'exécution du script en utilisant la combinaison de touches `Ctrl+C` dans la console d'exécution.

Les accumulateurs Le technique des accumulateurs est une méthode classique pour le calcul de sommes ou de produits de la forme suivante :

$$\sum_{i=1}^{n} f(i)$$

Un accumulateur est une variable dans laquelle on va ajouter la valeur $f(i)$ à chaque tour de boucle, où i sera un indice de boucle à incrémenter.

Exemple 11 - Accumulateurs

On cherche à calculer la somme suivante :

$$\sum_{i=1}^{n} i * cos(i)$$

avec $n = 100$.

```
a <- 0 # a est un accumulateur
n <- 100
i <- 1 # initialisation de l'indice de boucle
while( i<=n ) {
  # ICI a vaut sum_{k=1}^{i}{k*cos(k)}
  a <- a + i*cos(i)
  i <- i+1 # incrementation
}
```

De même que pour les indices de boucle, il faut bien penser à initialiser l'accumulateur. Dans ce cas où une somme sans terme étant nulle, on initialie à 0.
Le commentaire de la ligne 5 donne l'invariant de boucle pour un accumulateur, exprimé en fonction de i. Il s'agit de la somme partielle. Pour fournir un résultat correct, il faut bien veiller à ajouter le terme pour n, la condition est donc `i<=n`.

2.5.2 Les boucles for

D'un point de vue algorithmique, toute boucle `for` peut facilement se traduire en boucle `while`. Donc, les mêmes attentions doivent être apportées aux `for` que pour les `while`.
Dans le cas de boucles `for` qui utilise un intervalle du type `a:b`, les deux programmes ci-dessous sont équivalents. Vous noterez que dans le cas du `while`, il ne faut surtout pas oublier l'incrémentation de `i`, sinon le programme ne traitera que le premier élément à l'infini !

```
#INIT
for(i in a:b) {
 # DO i
}
```

```
#INIT
i <- a
while( i<=b ) {
 # DO i
 i <- i+1
}
```

Il est souvent utile d'utiliser des boucles `for` lors des parcours de collections d'objets : listes, matrices et de vecteurs. Il est également possible d'écrire des équivalences "typiques" entre les deux formes d'itération. On donne ci-dessous l'équivalence dans le cas du parcours d'une liste de valeurs.

```
#INIT
L <- list( ...)
for(i in L) {
 # DO i
}
```

```
#INIT
L <- list( ...)
i <- 1
while( i<=length(L) ) {
 # DO L(i)
 i <- i+1
}
```

Il faut bien voir dans ce dernier exemple d'équivalence que le `i` n'a pas le même usage. Supposons, par exemple, que la liste `L` contienne des chaînes de caractères. Dans le cas du `for`, le `i` est un **iterateur**, le `i` vaut le i-ème élément de la liste. `i` est ici une chaîne de caractères. Dans le cas du `while`, `i` est un indice utilisé pour repérer un élément dans la liste. C'est un nombre. Pour avoir un programme équivalent, il faut traiter l'élément `L(i)`.

2.6 Le branchement

2.6.1 Cas générique : instruction goto

Le branchement est une instruction d'algorithmique dont il faut limiter l'usage. Dans les langages procéduraux qui proposent la notion de fonction (*cf.* Chapitre 1), l'utilisation de branchement est la plupart du temps superflu, voire la marque d'une mauvaise programmation. À ma connaissance, le langage R ne propose pas d'instruction de branchement.
Classiquement, un langage utilise l'instruction **goto** ainsi que des *labels* pour faire des branchements. Le label donne une position dans le programme, l'instruction **goto** demande au programme d'aller directement à l'endroit du label désigné pour continuer l'exécution. L'exécution se comprend bien, il me semble. Dans l'exemple ci-dessous (qui n'est pas de R), il y a deux labels et deux instructions **goto**. Si le programme arrive à la ligne 6, alors on passe directement à la ligne 12, c'est à dire la fin du programme, en dehors de la boucle `for`.
Algorithmiquement, le programme est correct puisqu'il s'arrête bien. Je ne vais pas le démontrer ...

```
j<-0
label-debut:
j<-j+1
for(i = 1:100) {
  if( j>34 ) {
    goto label-fin
  }
  if( i==50 ) {
    goto label-debut
  }
}

print("affichage intermediaire")

label-fin:
print("c'est la fin")
```

Remarque 6 - Branchement conditionnel

Un branchement en lui même n'est pas très intéressant, c'est juste un saut dans le programme. En algorithmique, le branchement conditionnel est réellement intéressant. Comme on le voit dans l'exemple, le branchement – c'est-à-dire l'instruction de **goto** – est toujours associée à une condition (un test `if`).

Comme indiquer précédemment, les instructions ne sont pas nécessaires. C'est qu'il est possible d'écrire le même programme sans faire appel aux **goto**. Rien de plus facile ...
Dans l'exemple, on est confronté à deux cas bien distincts, et un troisième cas pourrait être rencontré :
– Un cas de bouclage (`label-debut`) : l'instruction de **goto** fait revenir en arrière dans le programme pour reboucler
– Un cas de saut dans l'execution (`label-fin`) : l'instruction de **goto** conduit la machine à avancer "plus vite que prévu" dans l'exécution du programme.
– un cas de renvoi à un bout de code antérieur : sans nécessairement avoir de bouclage, cela permettrait d'exécuter un bout de code placé autre part dans le programme (c'est le concept de procédure ou fonction du Chapitre 1.

Dans la suite de ce paragraphe, on présente une méthode un peu technique de suppression des `goto`. Si les détails ne vous sont pas évident, vous pouvez passer rapidement sur la preuve de la réelle équivalence entre les algorithmes.
Nous allons commencer par supprimer le `label-fin` grâce à une technique de `flag` (un drapeau en bon Français). un *flag* est une variable (généralement booléenne) qu'on va utiliser pour savoir si la condition assodiée au saut est vraie ou non. On va définir une variable **flag1** qui indiquera que $j > 34$ lorsqu'elle vaut `T`. On va donc sortir de la boucle lorsque **flag1**=`T`. Pour conserver l'équivalence [3], il faut aussi tenir compte de l'autre condition de sortie de boucle, à savoir $i > 100$. On se retrouve alors à devoir transformer le **for** en **while** pour combiner plusieurs conditions.
Vous noterez que :

1. `flag1` a été initialisé à `F` pour pouvoir rentrer dans la boucles
2. l'affichage intermédiaire entre la fin de la boucle et le label a été conditionné à la valeur du *flag*
3. toute la fin de la boucle a été placé dans un **else** car dans le cas d'une condition vérifiée, il faut aller directement en fin de boucle !

```
j<-0
label-debut:
j<-j+1
i<-1
flag1 <- FALSE
while(i <=100 && flag1!=TRUE) {
  if( j>34 ) {
    flag1 <- TRUE
  } else {
    if( i==50 ) {
      goto label-debut
    }
    i <- i+1
  }
}

if( flag1!=TRUE ) {
  print("affichage intermediaire")
}

print("c'est la fin")
```

Ensuite, on va supprimer `label-debut`. Comme indiqué précédemment, c'est un cas de bouclage. La solution va donc consisté à introduire une boucle supplémentaire dans le code. De même que précédemment, on va définir un *flag* nommé **flag2** qui sera vrai lorsque $i = 50$. On voit que le branchement à pour effet de sortir de la boucle existante, il faut donc ajouter une nouvelle condition de sortie de la boucle prenant en compte **flag2**. Ensuite, il faut "enrober" avec une autre boucle pour permettre le retour en arrière et tenir compte de **flag2** comme condition de sortie.
Maintenant qu'il n'y a plus de `goto`, je peux écrire du vrai langage R :

```
j<-0
flag2 <- FALSE
while( flag2!=TRUE ) {
  j<-j+1
  i<-1
  while(i <=100 && flag1!=TRUE && flag2!=TRUE) {
    if( j>34 ) {
      flag1 <- TRUE
    } else {
      if( i==50 ) {
        flag2 <- TRUE
      } else {
        i <- i+1
      }
    }
  }
}
```

3. ici, il s'agira plutôt d'une équivalence esthétique que fonctionnelle !!

```
if( flag1!=TRUE ) {
  print("affichage intermediaire")
}

print("c'est la fin")
```

Bon, je le reconnais, la transformation est un peu technique et finalement pas très instructive ... puisque c'est uniquement un exemple. Mais croyez bien qu'il y a toujours possibilité de traduire l'algorithme avec branchement en utilisant cette méthode de drapeau.
L'autre intérêt de l'exemple est bien de montrer cette stratégie algorithmique d'utilisation d'un *flag* pour retenir facilement la valeur d'un test précédemment effectué.

2.6.2 Cas du next et du break

Si on revient maintenant au premier algorithme avec les branchements, on est un peu déçu que le programme à écrire en R pour avoir quelque chose d'équivalent est plus long et plus difficilement lisible qu'avec les branchements.
Alors, certains branchements sont "acceptables"! Ce sont les branchements de modification de l'exécution d'une boucle :
– next permet de revenir directement au début de la boucle (en revenant au test du while)
– break permet d'intérrompre immédiatement l'exécution de la boucle et de sortir juste à la fin de celle-ci (sans refaire de test d'un while)
Dans notre exemple précédent, on a vu que la suppression d'un `goto` avait pour conséquence de mettre toute la fin de la boucle dans un else qui rend la lecture du code difficile. Le break permet de supprimer ce else.
Dans l'exemple ci-dessous, il faut surtout noter que le break (comme le next) ne sorte que de la boucle "la plus proche". Donc lorsqu'on tombe sur l'un des deux break, on retombe sur la ligne 16 (après le `}`) pour remonter à la ligne 3.

```
1  j<-0
2  flag2 <- FALSE
3  while( flag2!=TRUE ) {
4    j<-j+1
5    i<-1
6    while(i <=100 && flag1!=TRUE && flag2!=TRUE) {
7      if( j>34 ) {
8        flag1 <- TRUE
9        break
10     }
11     if( i==50 ) {
12       flag2 <- TRUE
13       break
14     }
15     i <- i+1
16   }
17 }
18
19 if( flag1!=TRUE ) {
20   print("affichage intermediaire")
21 }
22 print("c'est la fin")
```

Exemple 12 - Utilisation d'un break pour un tirage uniforme dans un intervalle

Soit $[a,b] \subset [0,1]$, on veut faire un programme qui tire uniformément un nombre dans $[a,b]$. La solution consiste à répéter un tirage uniforme sur $[0,1]$ et à sortir de la boucle lorsqu'on a une valeur intéressante !

```
repeat{
  val <- runif( 1 )
  # ici val est un nombre entre 0 et 1
  if ( val>=a || val <=b ) {
    break
  }
}
#ICI val est un nombre entre a et b
```

L'utilisation du next relève souvent de l'astuce pour gagner quelques lignes de codes ...

Exemple 13 - Utilisation d'un next un traitement conditionnel

On souhaite estimer la distance moyenne entre deux points, l'un tiré entre $[a,b] \subset [0,1]$ et l'autre tiré entre $[c,d] \subset [0,1]$. Certe, c'est un problème assez artificielle ...
On va avoir besoin de calculer une moyenne, donc on prend la bonne vieille technique de l'accumulateur, et on va appliquer une méthode de rejet : on simule facilement des données et si elles ne répondent pas à nos critères, on ne les prends pas en compte, sinon on les prend en compte dans le calcul de la moyenne.

```
nb <- 0 # premier accumulateur contenant le nombre d'exemples
somme <- 0 # second accumulateur des distances
while(nb<1000) {
  p1 <- runif( 1 )
  p2 <- runif( 1 )

  if ( p1>=a || p1<=b ) {
    next
  }
  # ICI p1 est un nombre entre a et b

  if ( p2>=c || p2<=d ) {
    next
  }
  # ICI p2 est un nombre entre c et d

  nb <- nb+1
  somme <- (p1-p2)*(p1-p2)
}
```

Bien évidemment, c'est méthode peut être longue pour atteindre les 1000 exemplaires nécessaire à sortir de la boucle alors qu' on savait facilement générer des nombres aléatoirement dans un intervalle, sans faire de rejet. Mais lorsque la condition est plus complexe, il peut être difficile d'arriver à un tel générateur. La méthode de rejet est alors très utile.

Les next et le break peuvent s'utiliser dans n'importe quelle boucle : for, while et repeat. Ce dernier cas doit nécessairement inclure un break pour éviter le bouclage infini (*cf.* Section 2.5.1).

2.7 Exemples

2.7.1 Jeu non-équilibré

On cherche à tester empiriquement l'effet de biais d'un jeu de dé virtuel. Le principe du jeu est de lancer un dé à **m**-faces numérotée de 1 à m. Si le résultat du dé est inférieur à $(m-1)/2$ (on biaise un peu ...) alors le Joueur 1 gagne 1 point, sinon c'est le Joueur 2 qui gagne 1 point. La partie est recommencée 1000 fois. Le joueur qui gagne sera celui qui aura le plus de points à la fin.

Dans l'énoncé du programme, on peut lire que "la partie de dé sera recommencée 1000 fois" : on comprend donc que le programme devra avoir l'allure suivante :

```
i<-0
while(i < 1000) {
  #
  # faire une partie de de
  #
  i <- i+1
}
```

Si on s'intéresse à la fin du programme, on sait que le joueur qui a gagné est celui qui a le plus gagné de partie de dé. Il va donc être nécessaire de gagner de compter le nombre de partie de dé en utilisant dans accumulateurs. Dans le programme ci-dessous à gauche, on définit deux accumulateurs **score1** et **score2**. Une autre solution , ci-dessous à droite, consisterait à utiliser un unique accumulateur qui retient le nombre de partie gagnée en plus par le joueur 1 sur le joueur 2. Ces deux algorithmes sont équivalents.

```
i<-0

#  initialisation  des scores des deux joueurs
score1 <- 0
score2 <- 0

while(i < 1000) {
  # faire une partie de de,
  # si le Joueur 1 gagne score1 est incremente
  # si le Joueur 2 gagne score2 est incremente

  i <- i+1
}

if( score1>score2 ) {
  print("Joueur 1 a gagne")
} if( score1>score2 ) {
  print("Joueur 2 a gagne")
} else {
  print("Egalite")
}
```

```
i<-0

#  initialisation  du score
scorediff <- 0

while(i < 1000) {
  # faire une partie de de,
  # si le Joueur 1 gagne scorediff est incremente
  # si le Joueur 2 gagne scorediff est decremente

  i <- i+1
}

if( scorediff >0 ) {
  print("Joueur 1 a gagne")
} ( scorediff <0 ) {
  print("Joueur 2 a gagne")
} else {
  print("Egalite")
}
```

Intéressons nous maintenant à la partie de dé isolément de cette trame générale du programme. Pour simuler le jeu biaisé, on peut écrire le programme ci-dessous :

```
m <- 100
# generation d'un nombre entier  aleatoire
# uniformement entre 1 et m.
x <- 1 + floor( runif(1) * (m-1) )

# on regarde qui a gagne
if( x < (m-1)/2) {
  # ICI x < (m-1)/2)
  score1 <- score1 + 1
} else {
  # ICI x >= (m-1)/2)
  score2 <- score2 + 1
}
```

Ce programme peut remplacer les commentaires dans la trame générale du programme ci-dessus. On se rend compte facilement que le `m` n'a pas besoin d'être défini dans la boucle puisqu'il ne change jamais de valeur. On le place donc en dehors de la boucle.

```
m <- 100
i<-0

#  initialisation  des scores des deux joueurs
score1 <- 0
score2 <- 0
while(i < 1000) {
  # generation d'un nombre entier  aleatoire
  # uniformement entre 1 et m.
  x <- 1 + floor( runif(1) * (m-1) )

  # on regarde qui a gagne
  if( x < (m-1)/2) {
    # ICI x < (m-1)/2)
    score1 <- score1 + 1
  } else {
    # ICI x >= (m-1)/2)
    score2 <- score2 + 1
  }
  i <- i+1
}

if( score1>score2 ) {
  print("Joueur 1 a gagne")
} if( score1>score2 ) {
  print("Joueur 2 a gagne")
} else {
  print("Egalite")
}
```

– `score1` et `score2` sont des accumulateurs
– `i` est un indice de boucle qui permet de compter le nombre de partie qui sont jouées

3 Conditions

La manipulation des conditions est ... à faire !

4 Exercices

4.1 Manipulation de variables scalaire numériques

Exercice 1 *Voici, à un moment donné, la configuration des variables, toutes de type entier :*

```
x <- 12
y <- 19
z <- 14
w <- 19
```

Question a) Quelle sera la situation après l'exécution de (pour chaque nouvelle question on repart de l'état ci-dessus)

```
x <- y
y <- x
```

```
y <- x
x <- y
```

```
z <- x + y
y <- x + y
x <- y
```

```
y <- x
z <- y
w <- z
x <- w
```

Question b) Écrire le (petit) algorithme qui fera en sorte que l'ordinateur échange les contenus des variables x et y.

Exercice 2 (Permutation circulaire de trois variables) *Écrire un (petit) programme qui fera faire à l'ordinateur une permutation circulaire des variables x, y et z. C'est-à-dire que le contenu de x doit passer dans y, celui de y dans z et celui de z dans x.*

4.2 Conditions

Exercice 3 (Les conditions)

Question a) Pour les deux programmes ci-dessous, donner l'affichage du programme pour les valeurs de i ci-dessous

- `i<--1`
- `i<-2`
- `i<-4`
- `i<-6`

```
if( i>0 ) {
  cat("i ", i, " est fitisop \n");
} else if ( i<0 ) {
  cat("i ", i, " est fitagen \n");
} else {
  cat("lun\n");
}
```

```
j<-10;
if( i < 4) {
  if( j > 4) {
    print("ici~?");
  }
} else {
  if( j > 4 ) {
    print("la?");
  } else {
    print("ou la?");
  }
}
```

Exercice 4 (Conditions (⋆)) *On vous propose de comprendre le programme ci-dessous. La variables* `x`, `y`, `z` *et* `w` *sont des* numeric.

```
#ICI on ne sait rien sur l'ordre entre les variables

if ( x > y ) {
  w<-x
  x<-y
  y<-w
}
#ICI ...

if ( y > z ) {
  w<-y
  y<-z
  z<-w
}
#ICI ...

if ( x > y ) {
  w<-x
  x<-y
  y<-w
}
#ICI ...
print(x); print(y); print(z)
```

Question a) En imaginant que les variables ont été initialisées comme ci-dessous, indiquer ce qu'affichera le programme

```
x<-23.4
y<-120
z<-22.3
```

Question b) En vous intéressant aux ordres entre les différentes variables `x`, `y`, `z` et `w`, compléter les trois lignes de commentaires du programme ci-dessous.

Question c) Conclure sur ce que fait le programme ?

Exercice 5 (Inutile !) *Compléter les commentaires du code ci-dessous et supprimer les parties de code inutiles : soit parce qu'elles ne font rien, soit parce qu'elles sont inatténiables.*

```
n<-10
for (i in 1:n) {
    #ICI i est une valeur entre 1 et n

    if (i %% 2 == 1) {
      next
    }
    #ICI i ...

    if ( i%%3 == 0) {
      cat("divisible par trois\n")
    }

    if ( i > 45) {
      #ICI ...
```

```
      break
    } else {
      cat(i, "\n")
    }
    #ICI i ...

    if ( i>56 ) {
      cat ("
    }
  }
```

4.3 Boucles

Exercice 6 (Tableau d'évolution de variables)

```
prod <- 0
a <- 3
b <- scan()
if (b>0) {
  i <- 0
  while(i<b) {
    prod <- prod + a
    i <- i +1
  }
print prod
}
```

Exercice 7 (Boucle `for`)

```
j<-j0
d<-d0
while(j >= 1 ) {
  #ICI j vaut ...
  #    d vaut ...
  d<-d/2
  j<-j/2
}

#ICI j vaut ...
#    d vaut ...

cat("result : d=",d, "\n")
```

Question a) Pour les valeurs de **j0** et **d0** suivantes, indiquer quel sera l'affichage du programme ci-dessus

```
j0 <- 16
d0 <- 200
```

Question b) Compléter les commentaires pour exprimer les valeurs de j en fonction de $j0$, et de d en fonction de j, $d0$ et $j0$

Question c) Que fait le programme ?

Exercice 8 (Boucle `while` et fonctions mathématiques)

```
f <- 81.0
nbTours <- 0

while( f > 5 ) {
  # ICI f vaut ...
  f <- sqrt(f)
  nbTours<-nbTours+1
}
print (nbTours)
```

La fonction `sqrt` *correspond au calcul de la racine carrée de* `f`*. On rappelle que* $\sqrt{x} = x^{\frac{1}{2}}$.

Question a) Compléter le commentaire en exprimant la valeur de `f` dans la boucle en fonction de 81 et de `nbTours`.

Question b) En déduire analytiquement, l'affichage du programme

Question c) Mêmes questions en remplaçant la condition du `while` par `f>0`

Exercice 9 (Boucle à rebours) *Écrire un programme capable d'afficher un compte à rebours. C'est-à-dire qui affiche successivement 100, 99, 98, etc.*

Exercice 10 (Rallye mathématique 2010 (⋆⋆)) *L'exercice 10 de Second Rallye mathématique 2010 de la Réunion proposait d'étudier le programme ci-dessous.*

```
while(A!=B) {
  if ((A>B) {
    A<-A-B
  } else {
    B<-B-A
  }
}
```

Quel est la valeur de `A` *à la fin du programme ?*

Question a) Tester l'algorithme avec `A<-42` et `B<-60`

Question b) "d divise A" est il un invariant de boucle ?

Question c) Montrer que "d divise A et d divise B" est un invariant de boucle.

Question d) Montrer que $d = pgcd(A, B)$ est un invariant de boucle.

Exercice 11 (Table de multiplication) *Écrire un algorithme qui demande un nombre de départ, et qui ensuite écrit la table de multiplication de ce nombre, présentée comme suit (cas où l'utilisateur entre le nombre 7) :*

```
Table de 7 :
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
...
7 x 10 = 70
```

Exercice 12 (Factorielle itérative) *Écrire un algorithme qui demande un nombre entier à l'utilisateur, qui calcule et affiche sa factorielle.*
Rappel : la factorielle de 8, notée $8!$*, vaut* $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8$.

Exercice 13 (Projet Euler - Problème 1) *If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.*
Find the sum of all the multiples of 3 or 5 below 1000.

Exercice 14 (Projet Euler - Problème 6) *The sum of the squares of the first ten natural numbers is,* $1^2 + 2^2 + ... + 10^2 = 385$.
The square of the sum of the first ten natural numbers is, $(1 + 2 + ... + 10)^2 = 552 = 3025$.
Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is $3025 - 385 = 2640$.

Question a) Find the difference between the sum of the squares of the first n natural numbers and the square of the sum.

Question b) (⋆) Dessiner une courbe qui donne, en fonction de n, la différence entre la somme des carrées de n premiers entiers avec le carrée de la somme des n premier entiers (fonction `plot`).

4.3.1 Interruption de boucles

Exercice 15 (Compréhension) *Indiquer quel sera l'affichage du programme suivant :*

```
n<-50
for (i in 1:n) {
  if (i %% 2 == 0) {
    next
  }
  #ICI ... (que dire sur i ?)

  if ( i %% 3 ==0) {
    #ICI ...
    break
  }
  #ICI ... (que dire sur i ?)
  cat(i, "\n")
}
```

Aide : *On rappelle que l'opérateur* `%%` *calcule le reste de la division. Le premier test permet donc de tester simplement si un nombre est pair ou non.*

Exercice 16 (Boucles infinies ?)

Question a) Indiquer ce que fait le programme ci-dessous ?

```
while( T ) {
  cat("tour de boucle\n")
}
```

Question b) Même question pour les deux programmes ci-dessous ?

```
i<-0
while( T ) {
  cat("tour de boucle\n")
  if (i>10) {
    break
  }
  i<-i+1
}
```

Question c) Modifier le programme précédent en supprimant le `break`

Exercice 17 (Projet Euler - Problème 9 (⋆)) *A Pythagorean triplet is a set of three natural numbers,* $a < b < c$*, for which,* $a^2 + b^2 = c^2$
There exists exactly one Pythagorean triplet for which $a + b + c = 1000$*. Find the product* $a \times b \times c$*.*

4.3.2 Calcul de séries numériques

Exercice 18 (Somme des n premiers entiers) *Le programme ci-dessous permet de calculer la somme des* n *premiers entiers :*

$$\sum_{i=1}^{n} i$$

```
S<-0
n<-100
i<-0
while( i<=n ) {
  # ICI S vaut ...
  S <- S + i
  i <- i + 1
}
# ICI S vaut ...
print(S)
```

Question a) Compléter les commentaires par des expressions mathématiques dépendantes de i et n.

Question b) Transformer la boucle `while` du programme avec un `for`.

Question c) Modifier le programme pour calculer les sommes suivantes :
- *la somme des n premiers carrés :* $\sum_{i=1}^{n} i^2$
- *somme des n premiers entiers pairs (plusieurs solutions possibles),*
- *somme des n premiers entiers impairs (plusieurs solutions possibles).*

Question d) Modifier le programme pour calculer le produit des n premiers entiers.

Exercice 19 (Calcul de π)

Question a) En utilisant la formule ci-dessous, écrire une programme qui calcule une valeur approchée de π

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

Question b) Modifier le programme pour afficher à chaque itération la valeur de l'accumulateur

Question c) Modifier le programme pour qu'il affiche la différence avec la valeur "réelle" de π *(Aide : la variable* `pi` *est définie par défaut dans R.)*

4.4 Entrées-sorties

Exercice 20 (Maximum) *Écrire un programme qui récupère deux nombres et affiche le maximum des deux. NB : vous n'utiliserez pas la fonction* `max` *de R.*

Exercice 21 (Guessing game (⋆)) *Le jeu du guessing game est un jeu dans lequel le programme "pense" à un nombre entier entre 1 et 99 et c'est à l'utilisateur de le deviner avec le moins de tentative possible alors que la machine ne répond que par "plus grand", "plus petit" ou "gagné".*

Question a) Écrire le programme du *guessing game* avec les spécifications suivantes :
- *Initialement, le programme tire un nombre au hasard entre 0 et 100.*
- *Une invite annonce à l'utilisateur qu'il doit donner un nombre entre 0 et 100.*
- *À chaque proposition de l'utilisateur, le programme répond par "plus grand", "plus petit" ou "gagné". Si l'utilisateur n'a pas gagné, l'invite est réaffichée.*
- *Tant que le joueur n'a pas gagné, il doit fournir des propositions.*
- *"gagné" met fin au jeu, et alors on affiche à l'utilisateur le nombre de tentatives faites.*
- *Le nombre maximum de tentative*

Question b) Ajouter des commentaires permettant de montrer que l'algorithme termine
Pour tirer un nombre aléatoirement entre 1 et 99 (compris) vous pouvez utiliser l'instruction ci-dessous. `runif` *signifie "Random UNIForm". C'est une fonction qui tire aléatoirement une valeur (réelle) entre 0 et 1 de façon uniforme :*

```
number <- as.integer(runif(1)*99) + 1
```

Exercice 22 (Hi-fu-mi) *Le jeu hi-fu-mi (Pierre, papier, ciseaux) est un jeu qui oppose deux joueurs qui indique simultanément l'un des trois symboles hi, fu ou mi. Lorsque les deux joueurs donnent le même symbole alors il y a égalité. Si les symboles sont différents le gagnant dépend des symboles : Hi bat Mi, Fu bat Hi, Mi bat Fu.*
La stratégie optimale, au sens de la théorie des jeux, consiste à choisir les coups de façon aléatoire, de manière équiprobable. Nous allons donc facilement pouvoir écrire un programme optimal.

Question a) Écrire un programme qui permette à un utilisateur de jouer à hi-fu-mi avec l'ordinateur. L'ordinateur tirera aléatoirement un nombre 1, 2 ou 3. Pour vous simplifier la vie, vous pourrez coder Hi par 1, Fu par 2 et Mi par 3. *On n'attend pas du programme qu'il gère les erreurs de saisies de l'utilisateur.*

Question b) Écrire un programme qui demande à l'utilisateur un nombre, n, de partie, joue n parties de hi-fu-mi avec l'utilisateur et indique le gagnant à la fin (celui qui a gagné le plus de parties).

Question c) Modifier le programme pour pouvoir utiliser les textes `hi`, `fu` et `mi`. On traitera alors les problèmes de saisie de l'utilisateur.

CHAPITRE

III Algorithmique sur des tableaux

Très rapidement, surtout en R, l'utilisation des tableaux devient une nécessité par ce que l'automatisation invite à faire des traitements en séries. Il s'agit en fait d'une application assez immédiate des boucles qui permettent d'itérer des opérations similaires.
Dans la suite, on illustre des méthodes classiques de traitement de tableaux qu'il faut avoir croiser au moins une fois dans sa vie de programmeur ... On commencera le chapitre en présentant des méthodes de traitement des données d'un vecteur, *c.-à-d.* des tableaux unidimensionnels, ensuite, toujours sur des vecteurs, on présentera des algorithmes un peu plus complexes qui jouent non-seulement avec les données du vecteurs mais également avec les indices qui identifient les données traitées d'un vecteur. Finalement, nous reprendrons ces différentes méthodes en montrant qu'elles peuvent s'étendre à des tableaux multi-dimensionnels. On ne focalisera sur les tableaux à 2 dimensions, *c.-à-d.* les matrices.

! Attention ! - Raison pédagogique et raison pratique

Dans la suite, on sera amené à présenter des algorithmes sur des tableaux issue de l'algorithmique impérative. C'est bien l'objectif pédagogique visé. Néanmoins, **le lecteur ne doit pas être sans savoir que, dans la plupart des cas, les solutions proposées ne sont pas des bonnes solutions pratiques pour avoir un programme R efficace**. Ceci est dû à la nature vectorielle du langage R.
Dans l'exemple ci-dessous, on illutre un code permettant de sommer tous les éléments d'un vecteur `x`. Le temps de calcul de cette fonction est d'environ 3,1 sec. En utilisant la fonction `sum(x)`, le temps de calcul aurait été de 0.004 sec.

```
x <- rnorm(1e6)
s <- 0
for (i in 1:length(x)) {
  s <- s + x[i]
}
```

Exemple 14 - Génération d'un tableau d'entiers

Pour tester les algorithmes de ce chapitre, il est utile de savoir générer rapidement des tableaux d'entiers. L'exemple ci-dessous permet de générer un tableau de 100 entiers dont les valeurs sont entre 10 et 220 avec une loi uniforme.

```
T <- as.integer( 10+210*runif(100) )
```

L'exemple ci-dessous fait de même pour un tableau à deux dimensions de taille 10×7.

```
T <- matrix(as.integer(10+210*runif(70)),nrow=7)
```

1 Algorithmes de manipulation sur le contenu des tableaux

On commence ici par donner 4 grandes classes d'algorithmes permettant de manipuler les données d'un tableau. Dans tous les codes de cette section, sauf indication contraire, on considère que `T` est un vecteur de nombres réels. On rappelle que la longueur du vecteur est donnée par `length(T)`.

1.1 Traiter les données d'un tableau

Lorsqu'on a des données dans un tableau, il est intéressant de pouvoir les traiter les unes après les autres. Lorsque ces traitements sont indépendants les uns des autres, l'algorithme permettant d'appliquer le traitement

à chacun des éléments est assez simple.
Supposons qu'on souhaite faire un calcul sur les données (par exemple, le calcul de $cos(3*x)$), et qu'on souhaite afficher la valeur correspondante. L'opération élémentaire sera donc un bloc d'instruction comme dessous :

```
val <- cos(3*x)
cat( "pour ", x, " la valeur est ", val, "\n")
```

Une fois que le traitement a été défini, pour l'appliquer à tous les éléments de notre trableau **T**, il faut parcourir chacune des cases et effectuer le traitement sur l'élément de la case.
Le programme ci-dessous illustre un tel parcours à l'aide d'un **while**. Le tableau d'évolution de variable contient une colonne spécifique pour **T[i]**. la valeur affichée dans cette colonne doit être cohérente avec la valeur de **i** sur la même ligne. On peut constater que le parcourt du tableau s'écris simplement en insérant le traitement à faire dans la boucle en utilisant **T[i]** comme variable, le i-ème élément du tableau **T**.

```
1 i<-1
2 while( i<=length(T) ) {
3   # On effectue le traitement de la i-eme case de T
4   val <- cos(3* T[i] )
5   cat( "pour ", T[i], " la valeur est ", val, "\n")
6   i <- i+1
7 }
```

`T <- c(34, 56, 78)`

line	i	T[i]	i<length(T)
1	1	34	-
2	1	34	T
3-5	1	34	-
6	2	56	-
2	2	56	T
3-5	2	56	-
6	3	78	-
2	3	78	T
3-5	3	78	-
6	4	*	-
2	4	*	F
7	4	*	-

On propose ci-dessous deux autres versions du même programme. Le programme précédent comme celui ci-dessous à gauche utilise une variable **i** comme indice dans le tableau tandis que dans le programme ci-dessous à droite la variable **i** est utilisée comme un itérateur.
Faites attention à bien définir les traitements pour **toutes** les cases du tableau :
– les traitements commencent à la première case du tableau, numérotée 1.
– les traitements finissent à la dernière case du tableau, numérotée **length(T)**
– les indices doivent être des nombres entiers !

```
for (i in 1:length(T)) {
  # On effectue le traitement de la i-eme case de T
  val <- cos(3* T[i] )
  cat( "pour ", T[i], " la valeur est ", val, "\n")
}
```

```
for (i in T) {
  # On effectue le traitement de i, element de T
  val <- cos(3* i)
  cat( "pour ", i, " la valeur est ", val, "\n")
}
```

! Attention ! - indice et élément

Attention à ne pas confondre, dans sa tête et / ou dans un algorithme, l'indice d'un élément d'un tableau avec le contenu de cet élément

1.2 Remplissage de tableaux en utilisant des boucles

Le remplissage d'un tableau n'est pas beaucoup plus difficile. Il s'agit, en effet, d'une opération à effectuer à chaque case du tableau, indépendamment les unes des autres.
On commence par le cas trivial de l'initialisation d'un tableau avec une valeur constante **val**. Rien de compliqué !

```
val <- 34
for (i in 1:length(T)) {
  # On effectue le traitement de la i-eme case de T
  T[i] <- val
}
```

On continue par un exemple simple un peu plus intéressant pour l'algorithmique. On souhaite écrire un tableau de la forme suivante : `T=[4 8 16 32 64 ...]`. La première chose à faire est d'abstraire cette définition en une formule générique du terme d'un tableau. Un peu de réflexion nous amène à constater qu'il s'agit un suite géométrique ... en constatant qu'il s'agit de puissance de 2 et en prenant un soin particulier à regarder le premier terme de la suite, on peut donc écrire sous la forme

$$\forall i \geq 1,\ T(i) = 2^{i+1}$$

Voilà, on a fait le principal du travail puisque le programme qui en découle est le suivant :

```
for (i in 1:length(T)) {
  # On effectue le traitement de la i-eme case de T
  T[i] <- 2**(i-1)
}
```

1.3 Algorithme de recherche

Un algorithme de recherche est un algorithme qui sera en mesure de nous indiquer si un élément particulier (au moins) est présent dans un tableau. Le **critère de recherche** est le test à effectuer sur un élément `T[i]` pour savoir si c'est un élément d'intérêt.
On peut commencer par recherche si il existe un élément égale à 100, c'est un critère de recherche très simple. Pour cela, la méthode de l'algorithme ci-dessous consiste à tester chacun des éléments du tableau et à utiliser un *flag* **found** pour indiquer que l'élément a été trouvé ou non.

```
found <- FALSE
for (i in 1:length(T)) {
  # On effectue le traitement de la i-eme case de T
  if( T[i]==100) {
    found <- TRUE
  }
}
## ...
if( found ) {
  print("element trouve")
}
```

Dans la mesure où notre objectif programme consiste à savoir si le tableau comprend **au moins** un élément qui vaut 100. Il n'est pas nécessaire de continuer la recherche lorsqu'on en a trouvé un. Une optimisation du programme peut se faire des deux manières ci-dessous. La solution de droite ajoute simplement l'utilisation d'un `break` pour interrompre la boucle lorsqu'on a trouvé un élément intéressant.
La seconde solution propose de ne pas utiliser de drapeau. Sans drapeau, il est nécessaire de refaire le test `T[i]==100` pour savoir si l'élément a été trouvé. En effet, en fin de boucle, on sait que $i \leq length(T)$, si $i < length(T)$ cela signifie qu'on a trouvé un élément valant 100 dans le tableau, mais si $i = length(T)$ on ne peut rien dire : soit c'était la fin de la boucle, soit le dernier élément contenait 100. Pour être complet, il est nécessaire de refaire le test.

```
found <- FALSE
for (i in 1:length(T)) {
  # On effectue le traitement de la i-eme case de T
  if( T[i]==100) {
    found <- TRUE
    break
  }
}
# ICI on est sorti soit en fin de boucle, soit par le
    break
if( found ) {
  print("element trouve")
}
```

```
for (i in 1:length(T)) {
  # On effectue le traitement de la i-eme case de T
  if( T[i]==100) {
    break
  }
}
# ICI on est sorti soit en fin de boucle, soit par le
    break
if( T[i]==100 ) {
  print("element trouve")
}
```

1.4 Méthodes de recherche séquentielle avec mémoire

Une méthode de recherche avec mémoire consiste à recherche un élément avec un critère qui dépend du parcours effectué. L'exemple typique de ce type de recherche est l'identification de la valeur maximale dans un tableau.
Ce qu'on sait faire pour le moment, c'est regarder case par case les éléments du tableau. La stratégie qu'on va utiliser pour trouver le maximum du tableau implique de regarder individuellement chacune des cases. En effet, si je les regarde de manière individuelle, pour une case données, la case `i`, on peut vérifier qu'il n'existe aucune autre valeur dans le tableau qui soit plus grande que la case `i` ...
En fait, c'est exactement comme si on faisait une recherche simple en en utilisant comme critère `T[i]<x`.
Regardons ce que cela donne pour la case `i` :

```
#ICI i est un indice du tableau T correspondant a la case traitee
ismax <- TRUE #flag indiquant si i est le max~!
for( j in 1:length(T) ) {
  if( i==j ) {
    #On ne traite pas la case i
```

```
    continue
  }

  if( T[i]<T[j] ) {
    ismax <- FALSE
    break
  }
}
#ICI ismax indique si T[i] est la valeur max du tableau T
```

Et ceci, il faut le faire à toutes les cases, éventuellement en s'arrêtant dès qu'on est tombé sur le maximum [1]. C'est bien compliqué pour une tâche si simple ...
Es-ce qu'on traite réellement les cases individuellement ? ? non, pas vraiment ... on les regarde plutôt de séquenciellement, les unes après les autres. Lorsqu'on traite la case `i`, on a en fait traité toutes les cases $j \in [1, i-1]$. On a donc calculé la valeur maximale pour le sous-tableau avec i. L'idée va donc être de dire qu'on va retenir (mettre en mémoire) la valeur maximale calculée pour le sous-tableau et regarder si la valeur de la case `i` est au-dessus de ce max. Si tel est le cas, le maximum pour le sous tableau $[1, i]$ est la valeur de la case `i` sinon, c'est toujours la valeurs antérieure.

```
max <- T[1] # initialisation du max
for(i in 1:length(T) ) {
  #ICI max est la valeur maximale pour le sous-tableau [1, i-1]
  if( T[i]>max ) {
    # nouveau maximum
    max <- T[i]
  }
  #ICI max est la valeur maximale pour le sous-tableau [1, i]
}
```

Le principe générale sur lequel s'appuie cet algorithme est l'utilisation d'une mémoire contenant le résultat du traitement sur le sous-tableau qui a été traité jusqu'au traitement de la case `i`. Une fois ce principe compris, l'algorithme en lui même est relativement simple.

Remarque 7 - Initialisation de la variable `max`

La variable a été initialisée avec la première valeur du tableau `T`. C'est en effet la bonne manière de faire ici. Et, en particulier, aucune valeur entière constante n'aurait pu être utilisée (0, 34, -10000), car on n'a aucune hypothèse sur le domaine des valeurs que peuvent prendre les éléments de `T`.
En initialisant avec `T[1]` on n'aurait pu ne pas traiter le premier élément du tableau et commencer à 2.
L'autre solution aurait été d'initialiser `max` avec la valeur $-$`Inf`. Dans tous les cas, il fallait préter une attention très particulière à l'initialisation.

2 Algorithmes avec manipulation des indices

Les indices sont utilisés pour identifier un élément d'un tableau. Dans la section précédente, les indices sont simplement utilisés dans la définition des limites de la boucle et pour faire référence à l'élément du tableau correspondant. Dans tout ce qui a été fait, il y a eu une confusion importante qui est possible entre l'indice et l'itérateur. L'itérateur est un élément du tableau, spécifique à ce dernier. L'indice ce n'est qu'un nombre entier. On lui attribut une signification d'indice dans les algorithmes précédents, mais en restant un simple entier, on peut continuer à les utiliser.
Dans cette section, nous tâcherons de décorréler l'indice de sa matrice.
Nous commencerons par regarder des indices qui servent dans plusieurs tableaux. Dans un second temps, nous regarderons des algorithmes qui s'intéressent à des opérations sur le voisinage d'un élément du tableau. Finalement, nous regardons des algorithmes qui utilisent les indices de tableaux de manière totalement générique.

Si les algorithmes précédents pouvaient facielement être réécris en R de manière plus efficace et plus concise en utilisant les fonctionnamlité de calcul matriciel supprimant l'utilisation de boucles, les algorithmes ci-dessous deviennent plus difficiles à concevoir avec R sans faire appel aux boucles.

2.1 Utiliser les indices pour traiter un voisinage (`i+1`, etc.)

Si on observe un tableau à une dimension en ligne, alors la case i à deux voisines : gauche et droite. Dans le cas général (*c.-à-d.* pas les bords), on peut donner leur indice en fonction de i. Celle de gauche à l'indice $i-1$ et celle de droite à l'indice $i+1$. Celle encore plus à gauche à l'indice $i+2$, etc.

1. je ne vais volontairement pas plus loin, on poursuivra cette exemple une fois qu'on aura vu les double-boucles.

Il est alors possible de définir des algorithmes qui tiennent compte du voisinage d'un élément.
Prenons l'exemple d'un algorithme qui cherche à savoir si un tableau est ordonné dans un ordre croissant, c'est-à-dire que la valeur de chaque case est inférieure à celle de la précédente. Exprimé le problème ainsi conduit à se faire intervenir une case et sa voisine de gauche. La première devant être plus grande que la seconde.
On va raisonner par récurrence pour faire notre algorithme en disant qu'un tableau est croissant jusqu'à i ssi :
– le tableau est croissant jusqu'à $i-1$
– $T[i] > T[i-1]$
En exprimant la stratégie comme ceci, on retrouve un parcours séquentiel de notre tableau.

```
isincreasing <- TRUE
  # ICI le tableau est croissante jusqu'a i-1
  if( T[i]<T[i-1] ) {
    isincreasing <- FALSE
    break
  }
  # ICI le tableau est croissante jusqu'a i
}
#ICI isincreasing est vraie si toute la matrice est croissante
```

Remarque 8 - Attention aux bords

Dans l'exemple ci-dessous, vous aurez peut être remarqué que l'indice `i` variait à partir de la valeur 2, et non 1 comme d'habitute. En effet, il est important de ne pas commencer à 1, sinon lors du premier passage dans la boucle, il sera difficile pour l'ordinateur de réaliser le test qui implique `T[i-1]` puisque si $i = 1$, `T[i-1]` n'existe pas : c'est un débordement de matrice.
Lors de l'utilisation d'expression impliquant le voisinage d'une case, il est donc important de faire très attention aux indices.

2.2 Un indice pour plusieurs tableaux

Dans ce section, nous allons nous intéresser à des opérations qui impliquent des éléments de deux matrices, mais avec un seul indice ... comme quoi l'indice ne sera pas associée à un tableau !
Le premier cas est très simple (et très inutile [2]), il s'agit de calculer la somme de deux vecteurs $\vec{s} = \vec{a} + \vec{b}$, on a lors $\forall i,\ s_i = a_i + b_i$. Cette opération ne sera possible que si a et b sont de la même longueur.

```
if( length(a)~!= length(b) ) {
  cat("Operation impossible\n")
} else {
  for( i in 1:length(a) ) {
    s[i] = a[i]+b[i]
  }
}
```

L'exemple est très simple puisqu'on fait des opérations entre des éléments de tableaux qui ont les mêmes indices.

2.3 Opérations sur les indices

Il est plus intéressant de regarder des algorithmes qui font intervenir des éléments de tableaux qui ne sont pas au même indice, mais pour lesquels il existe une relation qui peut être explicitée entre les indices.
On va prendre comme exemple l'inversion d'un tableau. C'est-à-dire qu'on souhaite remplir un tableau `T2` avec des éléments d'un tableau `T1` dans l'ordre inverse. Les deux tableaux doivent être de même longueur l. Dans ce cas, on sait que le premier élément de `T1` doit être le dernier de `T2`, *c.-à-d.* l'élément repéré par l'indice `l`, le second élément de `T1` doit ensuite être celui de `T2` à l'indice `l-1` ...etc. On en déduit la relation générale suivante. Un élément de `T2` à la case i doit être rempli avec l'élément à la position $l - i + 1$ de `T1`. Une fois que cette relation a été trouvée, l'algorithme qui en découle est simple : il s'agit simplement d'une parcours du tableau `T2` et pour chaque case, on affecte la case avec l'élément correspondant d'après notre formule. C'est le cas de l'algorithme ci-dessous, à gauche.
La relation étant symétrique, il est possible d'écrire un algorithme réalisant la même tâche mais pour lequel l'indice `i` correspond plus à un indice qui parcours la matrice `T1`.

2. l'addition des deux vecteurs peut se faire en utilisation l'opérateur +.

```
if ( length(T1)!= length(T2) ) {
  cat("Operation impossible\n")
} else {
  l <- length(T1)
  for( i in 1:l ) {
    T2[ i ] <- T1[l-i+1]
  }
}
```

```
if ( length(T1)!= length(T2) ) {
  cat("Operation impossible\n")
} else {
  l <- length(T1)
  for( i in 1:l ) {
    T2[ l-i+1 ] <- T1[i]
  }
}
```

Ce qu'il faut retenir de cet exemple, tient uniquement dans les lignes 6 des deux exemples précédents : il est possible d'écrire des opérations quelconques dans les crochets qui servent d'indices.

Remarque 9 - Typage à l'intérieur des crochets dans R

Les indices d'un tableau sont nécessairement des entiers. Mais, dans sa grande souplesse, R permet d'écrire des nombres `double` dans les indices. Il faut simplement savoir qu'une opération de coersion en entier (`as.integer`) est faite sur la valeur passée entre crochets.

! Attention ! - Attention aux débordements

Il est impératif de vérifier que la formule donnant l'indice d'un élément du tableau permet bien de rester dans les limites du tableau pour éviter les débordements. Ceci est d'autant plus important en R qu'il n'y aura aucune vérification qui sera faite avant l'exécution de votre code.

Exemple 15 - Parcours gauche/droite

L'idée de cet exemple est de couper une Gaussienne centrée au plus tôt.
L'objectif de cet exemple est de trouver un algorithme, qui permettent de traiter les éléments d'un tableau du centre vers l'extérieur en allant alternativement dans la partie gauche du tableau, puis dans la partie droite. Dès qu'on tombera sur une valeur du tableau `Tab` *qui soit inférieur à notre seuil, on arrêtera le parcours.*
On note l la longueur du tableau et on suppose que l est impair (dans le cas général, il faudrait faire en sorte d'ajuster les expressions en fonction de la longueur du tableau).
Ce problème revient donc à trouver une expression de l'indice de parcours en fonction d'une variable i qui varierait de 1 à $length(T)$.
L'alternative entre deux modes, me fait immédiatement penser à la parité d'un nombre. Lorsque i est pair, je vais à droite, et si i est impair, je vais à gauche ... À droite et à gauche de quoi ? ? Et bien du centre ... soit la case avec l'indice $\frac{l+1}{2}$ si l est impair.
De quoi ai-je besoin :
– de savoir si i est pair ou non : pour cela, j'utiliserai l'expression `i%%2` *est est nulle si i est pair.*
– l'alternative : j'utilise la formule $(-1)^x$, écrite `(-1)**x`*, qui donne est négatif si x est impair et positif sinon*
En arrangeant tout cela ... j'arrive à l'expression suivante : `(l+1)/2 + round(i/2)*(-1)**(i%%2)`
Le terme `(l+1)/2` *donne la position du milieu, et le reste de l'expression permet de donner la distance au centre alternativement vers la droite (positif) et vers la gauche (négatif). La fonction* `round` *calcule l'arrondie de la division.*
Regardons maintenant sur quelques exemples si l'expression proposée convient bien. On prend le cas de $l = 55$, avec $i = 1$, on calcule alors l'indice 28, c.-à-d. le centre du tableau. Pour $i = 2$, l'expression donne 29 : on ne tombe pas deux fois sur le centre ! C'était un risque avec l'utilisation du `round`*. Allons maintenant aux bords. Pour $i = 55$, on tombe sur la valeur 0 ! ! Ah, il y a un problème ... il faut alors modifier l'expression pour corriger l'erreur. En regardant d'autres exemples (53, 51, 49), on voit qu'il y a un problème dans les arrondis. On propose donc l'expression suivante :*
`(l+1)/2 + trunc(i/2)*(-1)**(i%%2)`
Après de nouvelles vérifications, on est confiant sur la validité de cette expression.
Le parcours du tableau devient simple, et on peut écrire l'algorithme souhaité ci-dessous. Pour simplifier la lecture, j'ai mis le calcul de la position dans le tableau dans une variable `pos`*.*

```
l <- length(Tab) # doit etre impair
seuil <- 0.9
found <- FALSE # flag pour la recherche
i <- 1 #indice
while( i<l ) {
  pos <- (l+1)/2 + trunc(i/2)*(-1)**(i%%2)
```

```
  if ( Tab[pos] < seuil ) {
    flag <- TRUE
    break
  }
  i <- i+1
}
#ICI si flag est vrai on a trouve un ligne de coupe, sinon il n'y a aucune valeur sous le seuil
```

2.4 Les tableaux d'indices

Bien, si vous m'avez suivi jusque là, on peut s'attaquer à une partie un peu plus difficile conceptuellement. Si jamais vous butez sur cette partie là, c'est quelque part normal. Forcez vous à essayez de bien la comprendre, et une fois que ce sera acquis, vous aurez fait un gros pas dans la programmation.
L'idée de cette section est de s'intéresser à des tableaux d'indices. Dans la mesure où les indices sont des variables comme les autres, il est possible également d'avoir des tableaux dans lequel à la place de mettre des données, on mette des indices qui font références à des données dans un autre tableau. Ceci n'est pas un artifice pour vous embêter ! C'est effectivement très courant en programmation. Pour cela, on va prendre un exemple un peu plus explicite pour en montrer l'intérêt.
Plutôt que de travailler sur un tableau **Tab** qui contiennent des données numériques simples, nous allons travailler sur un **data.frame** que nous pouvons voir comme un tableau de données complexes.
On commence par générer un **data.frame** de longueur **size** avec trois colonnes : **x**, **y** et **val** (une lettre entre A et D).

```
size <- 10
L4 <- LETTERS[1:4]
Tab <- data.frame(cbind(x=as.integer(5*rnorm(size)), y=as.integer(10*runif(size))), val=sample(L4, size, replace=TRUE))
```

Voici un exemple de tableau **Tab** générées :

```
    x y val
1   0 2   A
2   4 0   A
3   0 3   D
4  -7 2   B
5  -1 1   B
6  -9 5   A
7  -4 1   A
8   6 0   A
9  -2 5   A
10 -5 7   D
```

Si on souhaite maintenant sélectionner toutes les lignes correspondant à la lettre A, il est possible de construire un nouveau **data.frame** qui contiendra une copie des données pour ces éléments, mais ceci impliquerait de recopier de nouveau les données, et si il y avait des modifications à faire sur les données de ces lignes recopiées, elles ne seraient pas répercutées sur le tableau d'origine.
L'idée va donc être de repérer les lignes qui nous intéressent par leurs indices. Et comme il peut y avoir plusieurs lignes d'intérêt, on utilisera un tableau d'indice pour les sélectionner toute.
L'algorithme ci-dessous permet de construire un tableau d'indices des enregistrements pour lesquels **val** vaut **A**. Vous noterez qu'on utilise la propriété d'auto-extension des vecteurs R ! Ici, c'est une propriété très utile (mais en fait coûteuse en temps de calcul).

```
TabInd <- c() #creation d'un tableau d'indices vide
NbInd <- 0
for( i in 1:size ) {
  #ICI NbInd donne le nombre d'elements dans TabInd
  if( Tab[i,]$val == 'A' ) {
    TabInd[NbInd+1] <- i #ajout d'un element dans le tableau d'indices
    NbInd <- NbInd+1
  }
}
```

Le résultat obtenu pour le jeu de données illustré plus haut est le suivant :

```
> TabInd
[1] 1 2 6 7 8 9
```

Une fois qu'on dispose d'un tableaux d'indice, il est possible d'utiliser ce tableau pour faire des opérations sur les lignes des indices. Dans l'exemple ci-dessous, on continue avec l'exemple précédent en calculant les moyennes des `x` et `y` pour les éléments identifiés par le tableau d'indice **TabInd**.
On utilise la variable intermédiaire `pos` pour retenir

```
xmoy<-0
ymoy<-0
for( i in 1:NbInd ) {
  pos <- TabInd[i]
  xmoy <- xmoy+Tab[pos,]$x
  ymoy <- ymoy+Tab[pos,]$y
}
xmoy <- xmoy/NbInd
ymoy <- ymoy/NbInd
```

On obtient les résultats suivant sur le jeu de données d'exemple :

```
> xmoy; ymoy
[1] -0.8333333
[1] 2.166667
```

3 Tableaux à 2 dimensions : les matrices

3.1 Principe de la double-boucle

L'exemple ci-dessous illustre une double-boucle. Il s'agit d'une boucle sur les `j` qui est à l'intérieur d'une boucle sur l'indice `i`. L'utilisation des `while` permet de bien comprendre ce qui se passe. L'exécution de ce programme est illustré par le tableau d'évolution de variable sur la droite.
Le programme rentre dans la boucle la plus générale impliquant l'indice `i`. Et puis, on rentre ensuite dans la "petite" boucle de l'indice `j` qui boucle sur toutes ces valeurs. Une fois toutes les valeurs de `j` passée, on ressort de la petite boucle en ligne 8, et on incrémente `i`. On revient alors en ligne 2 pour boucler (grand boucle sur les `i`). Ah ! que se passe-t-il pour les `j` ... en bien comme à la ligne 3 on remet `j` à sa valeur initiale, on recommence à boucler sur les `j`. Es-ce que ça va s'arrêter ? ? Et bien oui, puisqu'à chaque fois qu'on a fait un tour de petite boucle, on incrémente `i`, et donc il y a un moment où le test `i<=3` devient faux.

```
1 i <- 1
2 while( i<=3 )
3   j <- 1
4   while( j<=2 ) {
5     cat(i, ", ", j,"\n")
6     j <- j+1
7   }
8   i <- i+1
9 }
```

En fait, ce que permet cet exemple, c'est de traiter à la ligne 5 toutes les combinaisons de valeurs (i, j), $i \in [1, 3]$, $j \in [1, 2]$. C'est donc un bout de code très approprié pour faire le parcours d'une matrice en deux dimensions puisqu'une case d'une matrice en deux dimensions est repérée par un couple d'indices ligne/colonne dont il faut traiter toutes les combinaisons.
L'exemple ci-dessous donne exactement la même exécution avec des `for` :

```
for( i=1:3 )
  for( j=1:2 ) {
    cat(i, ", ", j,"\n")
  }
}
```

line	i	j	i<=3	j<=2
1	1	?	-	-
2	1	?	T	-
3	1	1	-	-
4	1	1	-	T
5-6	1	2	-	-
4	1	2	-	T
5-6	1	3	-	-
4	1	3	-	F
7	1	3	-	-
8	2	3	-	-
2	2	3	T	-
3	2	1	-	-
4	2	1	-	T
5-6	2	2	-	-
4	2	2	-	T
5-6	2	3	-	-
4	2	3	-	F
7	2	3	-	-
8	3	3	-	-
2	3	3	T	-
3	3	1	-	-
4	3	1	-	T
5-6	3	2	-	-
4	3	2	-	T
5-6	3	3	-	-
4	3	3	-	F
7	3	3	-	-
8	4	3	-	-
2	4	3	F	-
9	4	3	-	-

3.2 Adaptation des algorithmes de recherche

Nous allons donc adapter l'algorithme de calcul d'un maximum d'une matrice **M**. En fait, il n'y a rien de bien compliqué puisqu'on a maintenant tous les éléments :
– On vient de voir comment parcourir toute une matrice grâce à une double-boucle.
– On a vu la méthode générale d'une recherche séquentielle avec mémoire pour trouver un max.

```
max <- M[1,1]
for( i = 1:nrow(M) )
  for( j = 1:ncol(M) )
    #Traitement de la case (i,j)
    if( max<M[i,j] ) {
      max <- M[i,j]
    }
  }
}
```

Tous les algorithmes de la section 1 pourraient être repris en les adaptant pour des matrices. Nous ne les détaillerons pas ici. D'un point de vue algorithmique, il n'y aurait pas grand chose de nouveau.
Pour les algorithmes de la section 2.2, on peut également traduire les algorithmes. Les algorithmes deviennent nécessairement un peu plus complexes puisque les indices sont doubles. En particulier, les voisinages d'une case d'une matrice sont plus complexes : si on prend les case en diagonale il y a 8 voisins, sinon il y a 6 voisins. De même les parcours "originaux" de la matrices peuvent être imaginés à l'infinie.

3.3 Généralisation aux `array`

En R, les `array` sont des tableaux de dimension multiple. Il est possible de généraliser la notion de double-boucle à des dimensions supérieures. À chaque nouvelle dimension, il est nécessaire d'imbriquer une boucle supplémentaire. Algorithmiquement, il n'y a pas grand chose d'intéressant là dedans !

Exemple 16 - Maximum d'un tableau à 4 dimensions

On peut très bien avoir un tableau **A** *à 4 dimensions. Dans ce cas, rien de plus simple que de traiter indépendamment les éléments du tableau. Pour repérer une case, nous avons besoin de 4 indices* (i, j, k, l)

```
max <- M[1,1,1,1]
for( i = 1:dim(M)[1] )
  for( j = 1:dim(M)[2] )
    for( k = 1:dim(M)[3] )
      for( l = 1:dim(M)[4] )
        #Traitement de la case (i,j,k,l)
        if( max<M[i,j,k,l] ) {
          max <- M[i,j,k,l]
        }
      }
    }
  }
}
```

3.4 Autres doubles-boucles

En fait, les double-boucles ne se croisent pas uniquement lorsqu'on traite des matrices, mais plutôt lorsqu'on traite des double-indices. Ce genre de situation apparaît également lorsqu'on traite des données unidimensionnelle.
L'exemple ci-dessous illustre un cas de double indice utilisé pour trouver le maximum d'un tableau de manière non séquentielle.

Exemple 17 - Recherche du maximum ... la suite (⋆)

Revenons maintenant à la première méthode de recherche d'un maximum. La stratégie était de regarder pour chaque case, si cette case était un maximum (hypothèse de traitement purement individualisé des cases) (cf. Section 1.4).
Finalisont maintenant l'algorithme en utilisant le bout de code déjà construit pour

```
1 for( i in 1:length(T) ) {
2   #ICI i est un indice du tableau T correspondant a la case traitee
```

```
3    ismax <- TRUE #flag indiquant si i est le max~!
4    for( j in 1:length(T) ) {
5      if( i==j ) {
6        #On ne traite pas la case i
7        continue
8      }
9
10     if( T[i]<T[j] ) {
11       ismax <- FALSE
12       break
13     }
14   }
15   #ICI ismax indique si T[i] est la valeur max du tableau T
16   if( ismax ) {
17     max <- T[i]
18     break
19   }
20 }
21 # ICI max est defini !
```

Une petite remarque sur le commentaire final ... On constate que `max` *n'est pas defini au debut de la boucle, mais il est defini dans la boucle et dans une condition. Le fait que sa définition soit dans une condition doit amener à penser qu'il est possible que la variable ne soit pas initialisée, et donc qu'elle n'existe pas à la dernière ligne du code. Hors, l'analyse du programme permet de conclure que si le tableau* `T` *n'est pas vide, alors l'exécution passera au moins une fois à la ligne 17 parce qu'il existe au moins un max dans le tableau. Dans le cas où le tableau est vide,* `max` *ne sera pas défini et il faudra faire attention aux erreurs ultérieures.*

4 Un peu de R

Pour la plupart des algorithmes qui ont été présentés, il est possible d'utiliser des opérations de calcul matriciels spécifiques à R. Ces méthodes sont toujours plus rapides a exécuter et à écrire, mais elles sont parfois difficiles à lire, et il peut être difficile de les maîtriser toutes.
Remplissage avec l'expression $T(i) = 2^{i+1}$:

```
> T <- 2**(2:11)
> T
 [1]    4    8   16   32   64  128  256  512 1024 2048
```

Recherche d'un maximum d'un vecteur :

```
> T <- as.integer(200*runif(100))
> max(T)
[1] 196
```

Pour recherche des positions d'un maximum d'un vecteur, on utilise la fonction `which`. Cette fonction indique les indices pour lesquels le test en argument de la fonction est vraie :

```
> which(T==max(T))
[1] 70 85
```

Pour inverser les éléments d'un vecteur :

```
T2 <- T1[ seq(length(T1), 1, -1) ]
```

Si on reprend la Section 2.4, le calcul de la moyenne des x et y pour les éléments qui ne contiennent `A` comme valeur *val* peut être simplement écrites ainsi :

```
mean(Tab$x[Tab$val=='A'])
mean(Tab$y[Tab$val=='A'])
```

Les opérations sur les indices sont moins génériques, il faut souvent passer par des astuces. Par exemple, pour savoir si un vecteur est ordonné en ordre croissant, il est préférable de regarder la différence entre deux vecteurs décalés. Avec la fonction `which`, on peut savoir s'il existe des éléments négatifs. Mais comme on se fiche de leur localisation, on peut se contenter de la fonction `sum`.

```
> T <- seq(1, 20, 2)
> which( (T[2:10]-T[1:9])<0 )
integer(0)
> sum( (T[2:10]-T[1:9])<0 )
[1] 0
> T <- 100*runif(200)
```

```
> sum( (T[2:length(T)]-T[1:(length(T)-1)])<0 )
[1] 99
```

Attention, dans les codes précédent, il manque à chaque fois le test pour la dernière case.

Remarque 10 - Difficultés d'écriture

Les fonctions sont détournés de leurs usages principaux. Il est souvent nécessaire de connaître très exactement ce que font les fonctions pour produire de tels codes. Pour moi, l'usage de ces fonctions tient souvent de l'astuce, de l'intuition, plus que d'une construction raisonnée générale. Il est beaucoup plus difficile de vous donner des démarches générales puisque cela reposera généralement sur une transformation de la tâche pour un usage détourné de fonctions matricielles.

5 Exercices

5.1 En vrac

Exercice 23 (Remplissage d'un tableau) *Pour chacune des questions suivante, on cherche à contruire un tableau de taille n avec des données particulières. Pour chaque question, trouver le schéma général des données et proposer une algorithme pour construire le tableau.*

Question a) Tableau avec les positions d'indice pair et de 1 sinon.

$$[1, 0, 1, 0, 1]$$

Question b) Tableau décalé

$$[2, 3, 4, 5, 6, 7]$$

Question c) Tableau inversé

$$[-10, -9, -8, -7, -6, -5]$$

Exercice 24 (Remplissage interactif) *Écrire un algorithme qui déclare un tableau de n notes, dont on fait ensuite saisir les valeurs par l'utilisateur.*
NB : Vous n'utiliserez que des fonctions de type `scan(n-1)` *pour les saisies clavier.*

Exercice 25 (le "schtroumpf" (⋆)) *Toujours à partir de deux tableaux précédemment saisis, écrivez un algorithme qui calcule le schtroumpf des deux tableaux d'entiers. Pour calculer le schtroumpf, il faut multiplier chaque élément du tableau T1 par chaque élément du tableau T2, et additionner le tout.*
Par exemple pour $T1 = [4, 8, 7, 12]$ *et* $T2 = [3, 6]$*, le Schtroumpf sera :*

$$3 * 4 + 3 * 8 + 3 * 7 + 3 * 12 + 6 * 4 + 6 * 8 + 6 * 7 + 6 * 12 = 279$$

Exercice 26 (Simplifications)

Question a) Que produit l'algorithme suivant ? (on pourra faire un tableau d'évolution de variables)

```
1  Nb <- rep(0,5)
2  i<-1
3  while(i<=5) {
4    Nb[i] <- i*i
5    i<-i+1
6  }
7  i<-1
8  while(i<=5) {
9    print Nb[i]
10   i<-i+1
11 }
```

Question b) Peut-on simplifier cet algorithme (i.e. l'écrire moins de lignes) avec le même fonctionnalité ?

Question c) Mêmes questions pour l'algorithme suivant

```
Nb <- rep(0,6)
N[6] <- 1
k<-6
while(k > 2) {
  N[k-1] <- N[k]+2
  k <- k-1
}
i <- 1
while(i<=6) {
  print N[i]
  i <- i+1
}
```

5.2 Algorithmes de recherches

Exercice 27 *Écrire une fonction qui recherche la dernière occurrence d'un entier* **elem** *d'un tableau d'entiers tab. L'élément et le tableau seront passés en paramètre de la fonction.*

Exercice 28 *Écrire un programme qui compte le nombre d'occurrence d'une valeur* **val** *dans un tableau d'entiers* **T**.

Exercice 29 *Écrire un programme qui compte le nombre d'occurrence d'une valeur* **val** *dans un tableau d'entiers* **T**. *Écrire un programme qui détermine si compte le nombre d'occurrences d'une valeur* **val** *dans un tableau* **T**.

Exercice 30 *Montrer que le programme ci-dessous permet de retrouver efficacement une élément e dans un* ***tableau ordonné*** *t. Vous completerez les commentaires et vous donnerez une signification à chacune des variables pour vous y aider.*

```
debut <- 0
fin <- n-1
trouve <- F
while( debut <= fin && !trouve) {
  # ...
  i <- (debut+fin)/2
  if (t[i] == e) {
    # ...
    trouve <- vrai
  else if (t[i] > e) {
    # ...
    fin <- i-1
  else
    # ...
    debut <- i+1
  }
}
# ...
if ( trouve ) {
  indice <- i
} else {
  indice <- -1
}
# ...
```

5.3 Opérations sur les indices

Exercice 31 (Plus longue sous-séquence uniforme (⋆⋆)) *Écrire un algorithme qui calcule la longueur de plus longue sous-séquence d'un vecteur composée d'une seul valeur. Dans l'exemple, ci-dessous, la plus longue sous-séquence est composée de 5 et elle est de longueur 4.*

```
[ 1, 2, 3, 3, 1, 2, 2, 5, 5, 5, 5, 4, 5, 4, 4, 4 ]
```

Exercice 32 (Parcours gauche/droite) *L'exemple 2.3 proposait un algorithme pour faire un parcours alterné "Gauche/Droite" à partir d'un tableau de longueur impair.*
Faire le parcours Gauche/droite dans le cas pair et proposer un algorithme valable pour n'importe quel taille de tableau. Vous chercherez à écrire un algorithme court, comportant le moins de répétition de code possible !

Exercice 33 (Suppression) *Écrire un algorithme qui recopie un tableau* **Tab** *sans prendre en compte les éléments de valeur e. Avec* $e = 3$*, la recopie de* $Tab = [4, 5, 3, 6, 3]$ *donnerait* $[4, 5, 6]$*.*

Exercice 34 *Écrire un programme qui utilise un tableau d'entiers* **tab** *(p.ex. saisi par l'utilisateur), et qui affiche un booléen :* **TRUE** *si* **tab** *est un palindrome,* **FALSE** *sinon. Un palindrome est un mot qui peut se lire dans les deux sens, par exemple "laval", avec un tableau d'entiers, un palindrome à la forme suivante* $[1, 2, 4, 2, 1]$ *ou* $[-1, 3, 5, 5, 3, -1]$*.*

Exercice 35 (Inversion d'un tableau) *Écrire une fonction qui inverse les éléments d'un tableau d'entiers tab. Par exemple* $[1, 2, 7, 4]$ *devient* $[4, 7, 2, 1]$*.*

Question a) Proposer par un algorithme utilisant un tableau intermédiaire

Question b) (⋆)Proposer une autre solution "en place". c'est-à-dire que l'algorithme ne doit utiliser qu'un nombre de "case" (variable ou élément d'un tableau) indépendant de la taille du tableau d'entrée. En particulier, il est interdit d'utiliser un tableau annexe de la même taille que le tableau d'entrée.

5.4 Tableaux d'indices

Exercice 36 *On reprend la même structures de* **data.frame** *que dans la section 2.4. On définit un tableau* **ValInteret** *qui comprend une liste de lettres.*
Déterminer la moyenne des **x** *et des* **y** *pour les enregistrements qui ont pour* **val** *l'un des éléments de* **ValInteret**.
NB : commencez par le faire en utilisant une boucle, puis essayer en utilisant la commande **which**.

Exercice 37 (max en colonne) *Écrire une programme qui calcule un vecteur* **v** *contenant les maximums de chacune des colonnes d'une matrice* **M**.

Exercice 38 (Localiser les maximaux d'un tableau) *Écrire un programme qui construit un tableau* **idx** *contenant les indices des maximums d'un tableau* **T**.

Exercice 39 (String matching (⋆)) *On se place dans un contexte de données génétique. Une séquence ADN peut se représenter sous la forme d'un grand tableau à une dimension.*
Une séquence ADN de longueur 100 peut être simulée de la manière suivante :

```
ADN <- sample( c('A', 'C', 'T', 'G'), 100, replace='true')
```

L'objectif de l'exercice est de regarder quelques tâches de recherche de sous-séquences dans la séquence d'ADN.

Question a) Recherche d'une sous-séquence rigide stricte
On cherche ici à retrouver toutes les sous-séquences SS, p.ex. `ATGG`*, dans la longue séquences ADN. Nous allons présenté l'algorithme "brute-force", c.-à-d. l'algorithme dont la stratégie est la plus simple : elle fait tout sans utiliser d'astuce algorithmiques. L'inconvient d'un tel algorithme est sa faible efficacité.*
La stratégie consiste à parcourir toutes les positions de la longue séquence ADN une à une et à chaque position. Pour chaque position i*, on regarde pour chacune des positions* j *de la sous-séquence SS, il s'agit de regarder si il y a correspondance entre l'élément de la sous-séquence et l'élément de la séquence. Si il n'y a pas de correspondance, alors cela ne sert à rien de continuer à tester la sous-séquence, il faut passer à la position suivante dans la séquence d'ADN. Lorsque tous les éléments de la sous-séquence ont été retenus, alors il faut retenir cette position.*
Faire un dessin illustrant ADN et SS en cours d'algorithme et indiquer les positions des deux cases dont on cherche à vérifier la correspondance.
En déduire l'algorithme permettant d'identifier toutes les sous-séquences rigides strictes.

Question b) Recherche d'une sous-séquence rigide non-stricte *On cherche maintenant des sous-séquences pour lesquels on autorise quelques erreurs entre la sous-séquence SS et la séquence. On note p le nombre d'erreurs autorisés (à fixer par l'utilisateur).*
Modifier l'algorithme précédent pour faire en sorte de trouver toutes les sous-séquences avec au maximum p erreurs.

Question c) Recherche d'une sous-séquence non-rigide stricte ($\star\star$) *On revient maintenant à des sous-séquences strictes. Cette fois-ci, on souhaite maintenant laisser la possibilité d'avoir des insertions entre les correspondances de SS dans l'ADN. On note q le saut maximum entre deux acides aminés correspondants. La valeur q sera déterminée par l'utilisateur.*
Modifier l'algorithme précédent pour extraire les sous-séquences strictes non-rigides.

CHAPITRE

IV Décomposition et fonctions

Le langage R est en plus d'être un langage de traitement hérite du fonctionnement des langages fonctionnelles comme le *scheme*. Les langages dits "fonctionnelles" utilisent la notion de fonction comme élément de base de la programmation (avant la boucle). On peut rapidement remarque que le langage R propose un très grand nombre de fonctions qui vont permettre de traiter les données. Pour les plus simples, se sont les fonctions `max`, `runif`, `seq`, etc.
Un programme est généralement un code long et compliqué fait à partir d'instructions rudimentaires. Une suite de 10 000 instructions rudimentaires serait incompréhensible. On constate également par la pratique que les programmes on souvent besoin de faire les mêmes choses (ou quasiment les mêmes choses). Il serait dommage de recopier 100 fois les mêmes lignes d'instructions : c'est consommateur en temps de développement, c'est générateur d'erreur et c'est compliqué à corriger (il faut penser à corriger une erreur pour les 100 occurrences du même bout de programme). Il est donc courant de "regrouper" des blocs d'instructions similaires sous la forme de fonctions : les fonctions sont elles même utilisées pour d'autres fonctions et au final, le programme principal sera composé de quelques fonctions et d'un programme principal en charge de faire quelques utilisations de fonctions.

Exemple 18 - Exemple de fonction

Supposons que vous souhaitiez calculer les intérêts du dépot annuel d'argent.
Si on note l'intérêt annuel i et D la somme déposée chaque année, alors la formule donnant le total votre compte en banque à la fin de n années est donné par :

$$D(1+i)^{n-1} + \ldots + D(1+i) + D = D\frac{(1+i)^n - 1}{i}.$$

Le code permettant de calculer ce montant final est le suivant :

```
D <- 2000
i <- 0.25
n <- 10
amount <- D * ( (i+1)**n -1 )/i
```

Dans l'absolue, vous souhaitez pourvoir calculer facilement ce montant dans votre programme pour différences valeurs de D, n et de i (par exemple pour comparer entre 10 ans et 20 ans). S'il est nécessaire de recopier systématiquement ces lignes, cela devient fastidieux.
Il est alors préférable de faire une fonction comme ci-dessous :

```
# Fonction pour le calcul d'un resultat de placement
#
# param D : depot annuel, nombre reel positif
# param i : interet , nombre entre 0 et 1 (0.05 par defaut)
# param n : nombre d'annee, nombre entier positif (10 par defaut)
# return le montant du compte a la fin des n annees
placement <- function(D, i=0.05, n=10)
{
  amount <- D * ( (i+1)**n -1 )/i
  return(amount)
}

# utilisation de la fonction
placement(2000, 0.25, 10)
placement(2000, 0.25, 20)
```

Le code ci-dessus illustre al définition et l'utilisation d'une fonction. La définition est accompagné d'un cartouche (commentaires au dessus de la fonction) qui décrit la fonctionnalité de celle-ci. Plus cette description sera précise (et concise) et mieux cela sera pour le programmeur qui utilisera votre fonction. Par exemple, l'utilisation d'un intérêt en pourcentage peut amener l'utilisateur à se demander si l'intérêt est une valeur entre 0 et 100 ou entre 0 et 1. En précisant cela, on évite un mauvais usage ultérieure et on facilite la compréhension du code.

L'intérêt d'une fonction est d'implémenter une "fonctionnalité" bien spécifiée à un seul endroit. À chaque fois que le programmeur veut utiliser cette fonctionnalité, il peut utiliser (faire appel) à cette fonction. De la sorte, le programmeur 1) économise du temps, par ce qu'il n'a pas à recoder la même fonctionnalité, 2) il évite des erreurs car moins il a de ligne de code à écrire moins il fera d'erreur, 3) il rend le code plus lisible, car tout le code n'est pas en vrac (décomposition logique) et 4) il facilie la correction/modification de son code. Si la fonction est effectivement utilisée plusieurs fois, la modification de la fonctionnalité n'impliquera la modification que d'un unique bout de code (la fonction) et garantira que la modification s'appliquera à chaque fois que cela était nécessaire.
Comme la plupart des langages, R permet au programmeur de définir ses propres fonctions pour décomposer son code selon les fonctionnalités qu'il veut utiliser dans son programme. Ceci permet au programmeur d'étendre le langage en proposant de nouvelles fonctions.

On voit ici apparaître l'une des très grosse difficulté de la programmation : comment décomposer son code ? La décomposition logique d'un programme constitue l'expertise principale d'un programmeur.

1 Fonctions

Une fonction est une sorte de boîte noire :
- **à l'extérieure**, elle est vue **comme 1 instruction** qui réalise une tâche de traitement de données (peu importe comment !),
- **à l'intérieur**, c'est **un (mini-)programme** qui implémente la sous-tâche de traitement de l'information.

1.1 Utilisation des fonctions

La définition d'une fonction et son utilisation nécessite d'avoir bien compris les méchanismes de passage de paramètres. Les paramètres peuvent être passés en utilisant leurs positions, soit en faisant explicitement appel au nom du paramètre (*cf.* polycopié de syntaxe du R). Dans cette section sur les fonctions nous utiliserons uniquement la référence positionnelle, sans paramètres optionnels. Le lecteur saura facilement adapté aux appels de fonctions avec paramètres nommés.
Quelques définitions
- paramètres formels : ce sont les paramètres donnés entre parenthèses lors de la déclaration de la fonction. Ces paramètres peuvent être vu comme des variables locales à la fonction. Elle seront initialisés lors de l'appel de fonction.
- paramètres effectifs :ce sont les paramètres qui sont données entre parenthèse lors de l'appel d'une fonction.

L'exemple ci-dessous illustre la définition d'une fonction et son utilisation à deux reprises.

```
1 mafonction <- function( a, b ) {
2   # corps de la  fonction
3   toto <- a+b*2
4   return(toto);
5 }
6
7 a <- 3
8 v <- mafonction(a, 56)
9 v <- mafonction(5, a)
```

line	a	v	a	b	toto
7	3	?	-	-	-
8	3	?	-	-	-
1	-	-	3	56	?
3	-	-	3	56	115
4	-	-	3	56	115
8	3	115	-	-	-
9	3	115	-	-	-
1	-	-	5	3	?
3	-	-	5	3	11
4	-	-	5	3	11
8	3	11	-	-	-

Le tableau d'évolution de variables fait apparaître deux contextes distincts : le context général dans lequel sont définies les variables **a** et **v**, et le contexte de la fonction dans lequel sont définies les variables **a** (ce n'est pas le même !), **b** et **toto**.
Contrairement à d'habitude, le début de l'exécution commence à la ligne 7, *c.-à-d.* la ligne de la première instruction du contexte général.
Lors de la ligne 8, le premier appel de fonction initialise les paramètres formels de la fonction avec les valeurs qui sont données dans la parenthèse de l'appel de fonction. On passe alors dans le contexte de la fonction en initialisant les variables en paramètres avec ces valeurs (correspondances positionnelles). La machine exécute alors les instructions de la fonction jusqu'au **return**.
L'instruction **return** à pour effet de sortir du contexte de la fonction et d'indiquer la valeur qui sera "retournée", ici, il s'agit de la valeur de la variable **toto**. On repasse alors dans le contexte général et on affecte la variable **v** avec la valeur "retournée". Cette étape se déroule de nouveau à la ligne 8.
Lors du second appel de fonction, ligne 9, on recommence la même opération. Deux choses me semblent importantes à noter. Tout d'abord, on remarque que lorsqu'on revient de nouveau dans le contexte de la fonction,

la variable **toto** n'a plus de valeur (noté ?). Les variables déclarées à l'intérieur d'une fonction sont "nouvelles" à chaque appel de fonction. Ensuite, on constate que la variable **a** du contexte général est utilisé en second paramètre, c'est-à-dire qu'elle initialise la variable **b** dans le contexte de la fonction. On se fiche totalement des noms des variables qui sont utilisés en paramètres effectifs.

Remarque 11 - Procédures

Généralement, les langages de programmation distinguent les procédures et les fonctions par le fait qu'une fonction "retourne" quelque chose. Dans le cas de R, il n'y a que des fonctions. Les procédures sont alors des fonctions qui retournent un vecteur vide (object appelé **NULL**).

1.1.1 Paramètres et retours : passage par valeurs

Il est très important de noter que les paramètres et le retour fonctionnent "par valeur"[1]. Lors de l'appel de fonction, il faut immaginer qu'une première étape consiste à remplacer l'intérieur des parenthèses par des valeurs (remplacement des variables par leurs valeurs ou calcul des opérations éventuelles).
En conséquence, il n'y a aucun "lien" entre les paramètres effectifs et les paramètres formels. Lors de l'exécution de la fonction, la modification d'un paramètre formel n'a aucune influence sur la valeur du paramètre effectif.

! Attention ! - Absence de typage des fonctions

L'absence de typage des fonctions n'impose rien au programmeur de la fonction en ce qui concerne son retour. Ceci est critique dans deux cas où le programmeur aura été inattentif :
- la fonction retourne des variables de types différents (par exemple, parfois un entier et parfois une matrice) : ceci est très dangeureux pour l'utilisateur de la fonction qui ne peut pas s'attendre à avoir à jongler avec différentes sorties de la fonction.
- la fonction ne retourne pas toujours quelques chose : ceci arrive plus fréquemment, lorsque le programmeur n'aura pas attentivement traiter tous les cas possibles. Si il n'y a pas de **return** alors la fonction va renvoyer **NULL**. Si l'utilisateur de la fonction s'attend à récupérer une valeur, il y a de forte chance que la suite de l'exécution plante.

Le programmeur d'une fonction R est le seul responsable du retour correct de sa fonction.

1.1.2 Commenter une fonction

Un cartouche correspond à la documentation d'une fonction pour des utilisateurs de celle-ci. En pratique, le cartouche est l'ensemble des commentaires sont placés juste aux d'une fonction.
Ce que doit préciser les commentaires de la fonction R :
- Une description générale
- les types attendus des paramètres et le type du retour de la fonction
- les préconditions : ensemble d'assertion qui doivent être vérifiée lors de l'appel de fonction. Les préconditions portent (principalement) sur les paramètres de la fonction
- les postcondictions : ensemble d'assertion que la fonction garantie à la fin de son exécution. Les post-conditions portent principalement sur la valeur de retour.

Les deux premiers éléments permettent aux programmeurs d'utiliser correctement votre fonction : il sait ce qu'elle fait et il sait comment l'utiliser. Les pré- et post-conditions permettent de raisonner formellement sur les programmes : ils peuvent être également intéressant pour les programmeur qui utilisent votre fonction.
Les commentaires que contiennent le cartouche doivent être : précis, exhaustifs et concis. L'usage de formules mathématiques dans les conditions sont tout à fait possible.
Il est important de faire l'effort de mettre un cartouche sur **toutes** vos fonctions. C'est un effort qui paye à long terme (pour la qualité du code et également pour la formation des programmeurs débutants). Rédiger un cartouche est une étape difficile, mais cela vous amènera à vous poser tout un tas de bonnes questions sur votre implémentation (avez vous bien pensez à tous les cas possibles, quels valeurs de paramètres faut-il interdire, ...).
Pour vous aider à la rédaction, *mettez vous à la place d'un programmeur qui veut utiliser votre fonction*, mais sans lire le code. Vous devez alors savoir comment utiliser la fonction et savoir exactement ce qu'elle fait !

1.2 Variables locales / Variables globales

Comme la majorité des langages de programmation, R comporte des concepts de variable locale et de variable globale (*cf.* Section 1.2).

1. Le passage de paramètre par "valeurs" s'oppose à des méthodes de passage de paramètres par "référence" ou par "pointeurs"

– Toute variable définie dans une fonction est locale à cette fonction, c'est-à-dire qu'elle n'apparaît pas dans l'espace de travail ni n'écrase une variable du même nom dans l'espace de travail (masquage).
– Il est possible de définir une variable dans l'espace de travail depuis une fonction avec l'opérateur d'affectation <<−. Il est très rare – et généralement non recommandé – de devoir recourir à de telles variables globales.
– On peut définir une fonction à l'intérieur d'une autre fonction. Cette fonction sera locale à la fonction dans laquelle elle est définie.

Le lecteur intéressé à en savoir plus pourra consulter les sections de la documentation de R portant sur la portée lexicale (lexical *scoping*). C'est un sujet important et intéressant, mais malheureusement trop avancé pour ce document d'introduction à la programmation en R.

1.3 Effets de bord

Les effets de bords désigne des actions que peut faire une fonction autre que le calcul de sa valeur de retour. Il existe deux exemples d'effets de bords classiques :
– les affichages qui peuvent être commandés à l'intérieur d'une fonction
– la modification d'une variable globale

Il semble qu'il ne soit pas possible d'utiliser des effets de bords sur des variables globales en R.

Exemple 19

```
1  y <- c(34, 45)
2
3  mafonction <- function( a, b ) {
4    toto <- a+b*2
5    y[1] <- toto
6    print (y)
7    return(toto);
8  }
9
10 v <- mafonction(3, 5)
11 print (y)
```

En plus de sa fonction de calcul, la fonction **mafonction** *effectue des affichages lorsqu'on l'appelle.*
Concernant la variable **y**, *on s'attendrait à ce que l'affichage de la variable à la fin du programme soit* 13 45, *mais la variable contient toujours* 34 45. *Les modifications qui sont faites sur* **y** *ne sont plus effective en dehors de ce contexte.*
NB : l'effet est le même en utilisant des listes.

2 Approche descendante

On se replace ici sur le problème de conception algorithmique d'un programme qui doit réaliser une tâche complexe. Pour trouver une solution algorithmique à ce problème, l'approche descendante invite à décomposer cette tâche, qu'on va dire de premier niveau, en tâches de second niveau. Cette première étape permet d'avoir un algorithme abstrait (dans le sens où il ne peut être exécuté par la machine). Dans un second temps, l'analyste programmeur s'intéresse aux tâches de second niveau pour en donner des algorithmes utilisant des tâches de troisième niveau etc. jusqu'à décrire les tâches à partir d'instructions élémentaires.
La devise de l'approche descendante est "diviser pour régner" : le problème est divisé est sous-partie qu'on sait bien faire (*régner*), et qu'on sait combiner en utilisant les structures de contrôle usuelles.
L'approche descendante est une démarche intellectuelle pour aider à construire des algorithmes.

Exemple 20 - Idée de décomposition

Par exemple, si le programme a construire consiste à trouver le nom de l'anté-antépénultième étudiant à partir d'une liste de notes (c.-à-d. le 4ème à partir de la fin), il peut être difficile d'imaginer un programme qui va réaliser cette tâche. Je vais donc décomposer la tâche en deux tâches successives (qui seront donc structurés par la structure de contrôle de séquence) : 1) je vais classer les étudiants par ordre croissant, puis 2) je vais chercher le 4ème étudiants à partir de la fin de la liste.
J'ai donc transformer une tâche complexes en deux tâches beaucoup plus simples.
L'idée de la décomposition de la tâche est difficile à trouver dans le cas général. Elle est d'autant plus facile que le programmeur aura en mémoire des tâches élémentaires qu'il saura facilement programmer. Ici, j'ai été dirigé par le fait que je pensais facile d'ordonner les étudiants par l'ordre des notes. Ensuite, l'extraction du 4eme à partir de la fin sera trivial.

Les avantages de l'approche descendante sont les suivants :
– ceci va rendre le programme **beaucoup plus lisible** que si il était entièrement écris dans un seul programme, et donc ce sera beaucoup plus facile d'y trouver des erreurs ou, pour autre programmeur, de comprendre ce programme.
– **les briques intermédiaires sont réutilisables pour d'autres programmes**. Si ces briques sont bien conçues, elles sont suffisamment génériques pour pouvoir être utilisées à plusieurs endroits dans vos programmes. Plutôt que de refaire à chaque fois le même bout de programme, mieux vaut le faire une fois bien, et le réutiliser ensuite. !
– le travail est plus facilement partageable entre plusieurs programmeurs. Si deux programmeurs se sont bien mis d'accord sur les **spécifications** des fonctions, peut importe lequel des programmeurs a effectivement réalisé la fonction, ils pourront utiliser le travail de l'autre sans se soucier de comment c'est fait.
– on facilite la correction des erreurs. Si une erreur c'est glissée dans un bout de code dont l'exécution se répète, alors il sera plus simple de corriger une fois ce bout de cod écris dans une fonction plutôt qu'à chaque fois qu'il a été écris dans un programme principal unique.

En programmation, les sous-tâches peuvent s'écrire sous la forme de fonctions. Chaque fonction réalise une sous-tâche. Dans le langage R, un ensemble de fonctions peut être regroupé au sein d'un package qui constitue une boîte à outilts (les fonctions) cohérente pour s'attaquer à un type de problème (programmation modulaire).

Un programme sera désormais composé comme un ensemble de fonctions qui peuvent s'appeler les unes les autres ainsi que d'un **programme principal** qui sera le point de départ de l'exécution du programme.

Exemple 21 - Décomposition : la mise en oeuvre

On reprend notre tâche d'identification du 4eme étudiant à partir de la fin dans le classement des notes. Nous avons donc besoin d'une fonction qui fera le classement des notes et d'une fonction que extrait le 4eme étudiants de notre liste ordonnées. Nous appellerons ces fonctions respectivement `sort.students` *et* `get.student`*. Une première version de notre programme se présentera sous la forme ci-dessous.*

```
# notes est une data.frame avec Nom et Note
# retourne un tableau de notes ordonnes
sort.students <-function( notes ) {
  # ...
}

# notes est une data.frame avec Nom et Note
# pre : notes est ordonne selon les notes
# retourne le nom de l'etudiant a la 4eme position a partir
# de la fin dans l'ordre des notes
get.student <- function( notes ) {
  # ...
}

##################################
# Debut du programme principal
DF <- data.frame(Nom = c("Pierre", "Jean", "Jacques", "Arnaud", "Sylvain", "Marte"), Note = c(12, 8, 10,16,14,13))

DFo <- sort.students( DF )
#ICI Dfo est le tableau de notes ordonnees

Etudiant <- get.student( DFo )
#ICI Etudiant est l'etudiant a la 4eme position a partir de la fin
print (Etudiant)
```

À ce niveau, on a réduit notre problème général à deux problèmes simples. Il est maintenant temps de s'attaquer à chacun de ces problèmes. Il est alors possible de se focaliser simplement sur une sous tâche. À ce niveau, le cartouche de la fonction doit être suffisant pour que le programmeur de la fonction sâche comment utiliser les paramètres de la fonction et quoi retourner.
Tout d'abord, pour la fonction de mise en ordre du tableau de note, je vais utiliser la fonction `order` *qui donne les indices ordonnées d'un vecteur.*

```
# notes est une data.frame avec Nom et Note
# retourne un tableau de notes ordonnes
sort.students <-function( notes ) {
  NotesOrdonnees <- DF[ order(notes$Note), ]
  return NotesOrdonnees
}
```

Ensuite, pour la fonction d'extraction du quatrième à partir de la fin, j'écris la fonction ci-dessous. Dans ce cas, on comprend que la pré-condition est très importante pour que le résultat de la fonction soit correcte.

```
# notes est une data.frame avec Nom et Note
# pre : notes est ordonne selon les notes
# retourne le nom de l'etudiant a la 4eme position a partir
# de la fin dans l'ordre des notes
get.student <- function( notes ) {
  return notes[nrow(notes)-4+1,]$Nom
}
```

Vous auriez très bien pu écrire des fonctions totalement différentes. Pour peu qu'elles respectent les spécifications des cartouches le programme principal aurait été correct.

L'autre intérêt des fonctions est de pousser les programmeurs à généraliser leurs tâches pour rendre les programmes plus "puissants" à très peu de frais. La généralisation des tâches est permise grâce à l'utilisation des paramètres d'une fonction qui peuvent être séparés entre, d'une part, des données à traiter et, d'autre part, des paramètres qui spécifient le comportement de la fonction.

Exemple 22 - Généralisation des fonctions

Dans le cas de notre fonction get.student *nous avons travaillé pour avoir une fonction qui retourne le 4eme à partir de la fin. Mais j'aurais très bien pu demander d'accéder au n-ème sans que le programme soit beaucoup plus difficile à écrire. L'intérêt de la généralisation ? Et bien la fonction pourra être utilisée dans plein d'autres cas que je n'aurai donc plus à traiter !*

```
# notes est une data.frame avec Nom et Note
# p est la position de l'etudiant recherchee
# pre : notes est ordonne selon les notes
# retourne le nom de l'etudiant a la p-eme position a partir
# de la fin dans l'ordre des notes
get.student <- function( notes, p ) {
  return notes[nrow(notes)-p+1,]$Nom
}
```

Le programme principal devient alors

```
# Debut du programme principal
DF <- data.frame(Nom = c("Pierre", "Jean", "Jacques", "Arnaud", "Sylvain", "Marte"), Note = c(12, 8, 10,16,14,13))

DFo <- sort.students( DF )
#ICI Dfo est le tableau de notes ordonnees

Etudiant <- get.student( DFo, 4 )
#ICI Etudiant est l'etudiant a la 4eme position a partir de la fin
print (Etudiant)
```

3 Récursivité

Une **fonction récursive** est une fonction qui s'appelle elle-même soit directement, soit indirectement.

3.1 Récursivité simple

Exemple 23

Prenons un exemple de calcul d'une factorielle. Pour rappel, la factorielle de n (notée n!) correspond à la formule suivante :

$$n! = n.(n-1).(n-2)...3.2.1$$

Récursivement, la factorielle peut s'écrire très simplement ainsi :

$$n! = \begin{cases} 1 & si\ n = 1 \\ n.(n-1)! & sinon \end{cases}$$

La fonction correspondante peut alors s'écrire ainsi :

```
1 # n : entier positif
2 # retourne un entier
```

```
3  Factorielle <- function(n) {
4    if (n>1) {
5      Fact <- n * Factorielle(n-1)
6    } else {
7      Fact <- 1
8    }
9    return( Fact )
10 }
11
12
13 print("Donnez un entier positif: ")
14 n <- scanf(n=1)
15 fn <- Factorielle(n);
16 cat("La factorielle de ", n, " est ",fn,".\n");
```

Definition 1 - Cas d'arrêt

Dans une fonction récursive, il doit exister une condition pour laquelle on arrête la récursion (sinon la récursion serait infinie). Cette condition définit le **cas terminal** ou **cas d'arrêt**.

Dans l'exemple, le cas $n = 1$ est appelé le **cas terminal**.
Le tableau suivant représente la séquences des appels de fonctions lorsque l'utilisateur a rentré le chiffre 4 :

Fonction/Contexte	Ligne de code	n	Fact	fn
principal	15 (avant affectation)	4		???
Factorielle(4)	4	4	???	
Factorielle(4)	5 (avant affectation)	4	???	
Factorielle(3)	4	3	???	
Factorielle(3)	5 (avant affectation)	3	???	
Factorielle(2)	4	2	???	
Factorielle(2)	5 (avant affectation)	2	???	
Factorielle(1)	4	1	???	
Factorielle(1)	7	1	1	
Factorielle(1)	9	1	1	
Factorielle(2)	5 (après affectation)	2	2	
Factorielle(2)	9	2	2	
Factorielle(3)	5 (après affectation)	3	6	
Factorielle(3)	9	3	6	
Factorielle(4)	5 (après affectation)	4	24	
Factorielle(4)	9	4	24	
main	15 (après affectation)	4		24

Prenons un second exemple avec la fonction puissance $r : x \to x^n$. Cette fonction peut être définie récursivement :

$$x^n = \begin{cases} 1 & si\ n = 0 \\ x.x^{n-1} & sinon \end{cases}$$

La fonction correspondant s'écrit ainsi :

```
# x : nombre reel
# n : entier >=1
# retourne un nombre reel
xn <- function(x, n)
{
  if (n==1) {
    return 1;
  } else {
    return( x * xn(x, n-1) )
  }
}
```

3.2 Autres formes de récursivité

3.2.1 Récursivité multiple

Une définition récursive peut contenir plus d'un appel récursif. Nous voulons calculer ici les combinaisons C_n^p en se servant de la relation de Pascal :

$$C_n^p = \begin{cases} 1 & si\ p = 0\ ou\ p = n, \\ C_{n-1}^p + C_{n-1}^{p-1} & sinon. \end{cases}$$

La fonction s'écrit alors ainsi :

```
# n, p entiers >0, p<=n
# retourne un entier
combinaison <- function(n, p)
{
  if( p==0 | p==n ) {
    return(1)
  } else {
    return( combinaison(n-1, p) + combinaison(n-1, p-1) )
  }
}
```

Les appels récursifs des fonctions devient alors assez complexe. Ici, tout ce passe bien parce qu'on sait qu'on ne tombera jamais deux fois sur le même calcul de combinaison, mais parfois ce peut être le cas. Auquel cas, les mêmes calculs sont répétés plusieurs fois.

3.2.2 Récursivité mutuelle

Des définitions sont dites mutuellement récursives si elles dépendent les unes des autres. Ça peut être le cas pour la définition de la parité :

$$pair(n) = \begin{cases} \text{vrai} & \text{si } n = 0 \\ impair(n-1) & \text{sinon} \end{cases}$$

et

$$impair(n) = \begin{cases} \text{vrai} & \text{si } n = 1 \\ pair(n-1) & \text{sinon} \end{cases}$$

Les fonctions vont alors de soit ... (*cf.* exercice 47).

3.2.3 Récursivité imbriquée

La fonction d'*Akermann* est définie comme suit :

$$A(m,n) = \begin{cases} n+1 & si\ m = 0 \\ A(m-1,1) & si\ m > 0\ et\ n = 0 \\ A(m-1, A(m,n-1)) & sinon \end{cases}$$

Dans ce cas, selon les mêmes principes de traduction que précédemment, on peut définir une fonction avec une récursion imbriquée ... c'est presque sans intérêt ! Mais ça a le mérite de montrer que la récursion peut se définir à toutes les sauces et que ça fonctionne très bien en informatique !

3.3 Principes et dangers de la récursivité

Principe et intérêt : ce sont les mêmes que ceux de la démonstration par récurrence en mathématiques. On doit avoir :
- un certain nombre de cas dont la résolution est connue, ces "cas simples" formeront les cas d'arrêt de la récursion ;
- un moyen de se ramener d'un cas "compliqué" à un cas "plus simple".

La récursivité permet d'écrire des algorithmes concis et élégants. Il faut noter que tout programme itératif peut être traduit sous une forme récursive.

Difficultés :
- la définition peut être dénuée de sens et donc difficilement compréhensible,
- il faut être sûr que l'on retombera toujours sur un cas connu, c'est-à-dire sur un cas d'arrêt ; il nous faut nous assurer que la fonction est complètement définie, c'est-à-dire, qu'elle est définie sur tout son domaine d'applications.

Moyen : existence d'un ordre strict tel que la suite des valeurs successives des arguments invoqués par la définition soit strictement monotone et finit toujours par atteindre une valeur pour laquelle la solution est explicitement définie.
L'algorithme ci-dessous teste si a est un diviseur de b.

```
# a, b : entiers >0
# retourne un entier
diviseur <- function(a, b)
{
  if(a<=0) {
    return(-1) # erreur
  } else {
    if( a>=b ) {
      return a==b
    } else {
      return( diviseur(a, b-a) )
    }
  }
}
```

La suite des valeurs b, $b-a$, $b-2a$, ... est strictement décroissante, car a est strictement positif, et on finit toujours par aboutir à un couple d'arguments (a, b) tel que $b-a$ soit négatif, car défini explicitement (cas d'arrêt). Cette méthode ne permet pas de traiter tous les cas :

```
# n : entier >=0
# retourne une entier
syracuse <-function(n)
{
  if( n==0 || n==1 )
    return( 1 )
  else {
    if( (n%2) == 0) {
      return( syracuse(n/2) )
    } else {
      return( syracuse(3*n+1) )
    }
  }
}
```

Remarque 12

Le résultat de cette fonction est toujours 1 pour tous les entiers (problème ouvert...)

3.4 Non-décidabilité de la terminaison

Cette section commence à être un peu plus poussée en terme d'algorithmique, puisqu'on se demande ici si il existe des moyens de déterminer automatiquement si un programme donné termine quand il est exécuté sur un jeu de données. On ne cherche pas à savoir quel sera le résultat, on veut simplement savoir si le programme boucle à l'infini ou bien s'il s'arrête à un moment donné (on dit qu'il "termine"). Cette question n'est pas anodine. Elle a été le sujet de forte controverse au début du XXeme siècle entre de nombreux mathématiciens (logiciens) en réponse à la question de savoir si les mathématiques pouvaient répondre à toutes les questions. De nombreux mathématiciens tels (Leibnitz ou ... TODO) étaient alors persuadés que les mathématiques étaient un outil qui permettait de répondre à toutes les questions (posées en langage mathématiques). Lorsque le jeune Godël puis B. Russel proposa que ce n'était pas le cas, c'est travaux furent très controversés.
Et bien, la réponse est ... NON ! Il existe des programmes pour lesquels on peut prouver qu'il sera imposiible de savoir si le programme termine ou non. La conséquence est également qu'il existe de énoncés mathématique dont on peut prouver qu'on ne pourra jamais prouvés qu'ils sont vrai ou faux.

Qu'est ce que ça veut dire ? ? Ça veut dire que vous ne pouvez pas démontrer (pour tous les algorithmes) si le programme s'arrête ou pas : vous ne pouvez pas montrer qu'il va s'arrêter, mais vous ne pouvez pas non plus démonter qu'il va tourner à l'infini ! Vous n'en saurez rien ... c'est indécidable [2] !
La démonstration de ce résultat étonnant, est une démonstration par l'absurde utilisant un raisonnement diagonal (proposé par Godël).
Supposons que la terminaison soit décidable, alors il existe un programme, nommé `termine`, qui vérifie la terminaison d'un programme, sur des données.
À partir de ce programme on conçoit le programme `Q` suivant (en pseudo-code) :

2. Il est également étonnant de noter que la classe des problèmes indécidables est "majoritaire" dans l'ensemble des problèmes "informatisables" !

```
bool Q()
{
  resultat = termine(Q, {})
  while( résultat==vrai ) {
    attendre 1 seconde
  }
  return(resultat);
}
```

Supposons que le programme `Q`, qui ne prend pas d'arguments, termine. Donc `termine(Q, {})` renvoie vrai, la deuxième instruction de `Q` boucle indéfiniment et `Q` ne termine pas. Il y a donc contradiction et le programme `Q` ne termine pas.
Donc, `termine(Q, {})` renvoie faux, la deuxième instruction de `Q` ne boucle pas, et le programme `Q` termine normalement. Il y a une nouvelle contradiction : par conséquent, il n'existe pas de programme tel que `termine`, et donc le problème de la terminaison n'est pas décidable.

4 Un peu de R : application d'une fonction sur un vecteur

Le langage R se fonde sur un paradigme de programmation fonctionnel, c'est-à-dire où tout est fonction. Le "vrai" programmeur R doit nécessairement être familier avec la notion de fonction, et donc naturellement avec l'usage de la récursivité.
Il reste néanmoins que c'est un langage fait pour traiter des tableaux de données. Dans le paradigme fonctionnel, l'usage des boucles n'est absolument pas recommendé[3]. On a ainsi pu voir (*cf.* introduction du Chapitre III) que l'utilisateur de boucle sur les tableaux donnaient de piètres performances. Alors comment s'en sort-on ?
En pratique, le langage R propose un jeu de fonction particulière de type apply (apply, lapply, sapply, tapply, mapply et vapply).

Exemple 24 - Exemple de la fonction apply()

Dans cet exemple, on cherche à calculer les valeurs moyennes des colonnes d'une matrice mat.
La méthode itérative pourrait ressembler à cela :

```
z <- vector()
for (i in 1:ncol(mat)) {
  z[i] <- mean(mat[, i])
}
```

En utilisant la fonction apply() *on va indiquer de réaliser la fonction* mean *sur tous les éléments de la matrice en itérant sur la seconde dimension :*

```
z <- apply(mat, 2, mean)
```

Le résultat des deux programmes est le même, et même si le premier programme est déjà concis, le second l'est encore plus. En règle générale, la seconde version sera également beaucoup plus rapide que la première.
En somme, il est généralement préférable d'utiliser ce type de fonctions lorsque vous programmer en R, contrairement à ce qu'on vous fait majoritairement faire dans ce cours.

4.1 Syntaxe de la fonction apply

La syntaxe et les arguments de la fonction apply sont les suivants

```
apply(X, MARGIN, FUN, ...)
```

Une fonction de ce type prend en argument :
- `X` : un collection de données (vector, list , array, ... en fonction de la fonction utilisée),
- `MARGIN` :la dimension du vecteur sur laquelle itérer, si `MARGIN`=1 la fonction est appliquée sur les lignes et si `MARGIN`=2, la fonction est appliquée sur les colonnes d'une matrice.
- `FUN` : une fonction à itérer sur les éléments du vecteur,
- `...` : des arguments optionnels à la fonction (optionnel),

3. On peut reprocher aux langage R de proposer des outils qui ne sont pas du tout performants. En permettant d'écrire des boucles, le programmeur peut se laisser aller à la facilité en programmant des boucles, alors qu'en se formant à des langages fonctionnels, il aurait de meilleurs qualités de programme.

Dans le cas où la fonction à appliquer comporte plusieurs arguments, le premier argument correspondra toujours à la n-ième colonne ou ligne de `X` (en fonction de `MARGIN`). Les autres arguments de la fonctions seront passés à la fin de la fonction **apply** (argument "...").

Exemple 25 - Exemple pour calculer une moyenne tronquée

```
apply(mat, 2, mean, trim = 0.5)
```

4.2 Autres fonctions

La fonction **tapply** permet d'effectuer des calculs sur les éléments d'un vecteur connaissant leur appartenance à des catégories (facteurs).

```
> fac <- as.factor(sample(LETTERS[1:5], size = 20, replace = T))
> fac
 [1] A E B D D B A E B E A C E E B B C E D A
Levels: A B C D E
> x <- runif(20, 0, 10)
> tapply(x, fac, sort)
25
$A
[1] 0.1422657 1.9218015 3.3894531 5.3228755
$B
[1] 0.5105884 3.2262803 4.7701483 5.2399521 6.6570511
$C
[1] 4.973641 9.694104
$D
[1] 4.526778 7.040531 7.704796
$E
[1] 0.4712470 0.7388998 1.4465574 4.1397842 7.9786786 9.3307780
```

La fonction **lapply** est réservée aux objets **list** ou **data.frame**. Au contraire de **apply**, **lapply** ne possède pas l'argument `MARGIN`. Sur un **data.frame** le calcul se fera sur les colonnes.

Exemple 26 - Un exemple avec la fonction is

L'application de cette function à chaque élément de la liste permet de connaître son type (sa "class").

```
> L1 <- list(A = rnorm(10), B = c(T, F), C = mat)
> lapply(L1, is)
$A
[1] "numeric" "vector"
$B
26
[1] "logical" "vector"
$C
[1] "matrix"
5.7
"structure" "array"
"vector"
"vector"
```

5 Exercices

5.1 En vrac

Il n'y a pas vraiment d'exercice spécifique pour les fonctions. L'apprentissage de la décomposition n'est pas l'objet de ce cours. Comme exercice, vous pouvez recoder les exercices précédents en utilisant des fonctions!! En particulier, tous les exercices nécessitant des données en entrée (saisie utilisateur) pourraient être réécris dans une fonction en remplaçant la saisie utilisateur par un paramètre d'une fonction.

Exercice 40 (Fonctions)

Question a) Écrire une fonction de conversion des Euros en Dollars (et réciproquement), une fonction de conversion de des Dollars en Yen.

Question b) Écrire une fonction de conversion des Euros en Yen (et réciproquement) en utilisant les fonctions précédentes

Question c) Écrire une fonction qui calcule l'aire d'un rectangle à partir de sa longueur et de sa largeur

Exercice 41 (Modèle de polution de Ricker) *Considérons que l'on veuille étudier le comportement d'un modèle non-linéaire, le modèle de Ricker défini par :*

$$N_{t+1} = N_t exp\left(r\left(1 - \frac{N_t}{K}\right)\right)$$

où N_t est la taille de la population à la génération t. Ce modèle est très utilisé en dynamique des populations, en particulier de poissons. On voudra a l'aide d'une fonction simuler ce modèle en fonction du taux de croissance r et de l'effectif initial de la population N_0 (la capacité du milieu K est couramment prise égale à 1 et cette valeur sera prise par défaut).

Question a) Construire une fonction **ricker** avec le profil ci-dessous calculant la population à toutes les générations de 0 à **time**. Le résultat sera retourné sous la forme d'un vecteur.

```
ricker <−function(nzero, r, K=1, time=100)
```

Question b) Utiliser votre fonction pour dessiner l'évolution de la polution en fonction des générations.

Question c) Écrire une fonction **ricker.draw** qui **fait la même chose** que **ricker** avec pour effet de bord

Exercice 42 (Calculer des entiers premiers (⋆)) *Un entier est premier s'il est divisible uniquement par 1 et par plus même. L'expressions "est divisible" signifie que le reste de la division est nulle (par exemple 9 est divisible par 3, mais pas par 4).*
L'objectif de cet exercice est d'extraire efficacement tous entiers premiers inférieurs à une valeur fixée n.

Question a) Fonction **estpremier** : implémenter une fonction qui prend en paramètre un entier n et qui teste si celui-ci est premier ou non en regardant si il existe un entier plus petit que lui qui le divise.

Question b) Écrire un programme faisant appel à votre fonction permettant de déterminer les entiers premiers inférieurs à p. Vous testerez des valeurs de p qui permettent de ne pas trop attendre.

Question c) Modifier la fonction en constatant que si un entier n ne peut pas être divisible par un entier strictement plus grand que $\frac{n}{2}$!

Question d) Tester de nouveaux les valeurs de p pour lesquels vous étiez bloqué par le temps de calcul.

Question e) Algorithme d'Erathostenes *Modifier la fonction pour implémenter l'algorithme d'Eratosthenes. L'algorithme reprend le même schéma que précédemment, mais au lieu de tester tous les nombres inférieurs à $\frac{n}{2}$, il part du principe qu'un entier est premier si il n'est pas divisible par un entier premier plus petit que $\frac{n}{2}$.*

5.2 Les codes

Dans cette partie, on s'intéresse à des algorithmes qui permettent de coder ou décoder un texte en clair. En plus de l'utilisation de fonctions, qui vous sont proposées, ces méthodes font intervenir des tableaux.
Quelques commandes utiles :

- `y <−LETTERS(1:26)` : vecteur contenant l'alphabet entier
- `y <−sample(1:26, 26)` : vecteur contenant la séquence 1 à 26 dans le désordre (sans répétition)
- `y <−sample(1:26, 26, replace=TRUE)` : vecteur contenant la séquence 1 à 26 dans le désordre (avec répétition)

Pour ces quelques exercices, vous pouvez vous mettre à deux groupes : un groupe qui écris un programme pour encoder un message et un groupe qui écris le programme pour décoder un message.
Avant de commencer les algorithmes de codage proprement dit, il est nécessaire de préparer les données pour les traiter. On a besoin de deux méthodes un peu "techniques" :
Construire un vecteur de lettres à partir d'un texte :

```
texte <- "Ceci est un texte secret"
vx <- substring(x, 1:nchar(x), 1:nchar(x))
```

Construire un texte à partir d'un vecteur de lettre :

```
t<-""
for(i in 1:length(vx)) {
  t<-paste(t,vx[i], sep="")
}
```

Exercice 43 (Code de César) *Le code de César permet d'encoder un texte en utilisant un alphabet encodé faisant correspondre une lettre en clair à une autre lettre chiffrée.*
Pour construire l'alphabet encodé, le code de César utilise un mot clef. Par exemple, prenons "Julius Caesar" comme mot-clef, et commencons par supprimer les espaces et les lettres qui sont répétées (JULISCAER). Utilisons la suite obtenue comme début de l'alphabet codé. Le reste de l'alphabet codé n'est que le résultat d'un simple enchaînement qui commence où finit le mot-clef, en omettant les lettres qui figurent déjà dans celle-ci. Ainsi, l'alphabet codé se présentera comme suit :

```
abcdefghijklmnopqrstuvwxyz
JULISCAERTVWXYZBDFGHKMNOPQ
```

Question a) Écrire une fonction qui ne conserve que les lettres (sans espace ni caractères spéciaux) à partir du vecteur de lettre.

Question b) Écrire une fonction qui encode une lettre à partir d'un alphabet codé

Question c) Écrire des fonctions qui encode ou décode un texte à partir d'un texte en clair et d'un alphabet codé

Question d) Écrire une fonction qui construit un alphabet chiffré à partir d'une mot-clef (⋆⋆)

Question e) Écrire des fonctions qui encode ou décode un texte à partir d'un texte en clair et d'un mot-clef.

Exercice 44 (Codage de Vigenère) *Le codage de Vigenère utilise un tableau qui représente les 26 alphabets circulairement permutés.*

```
abcdefghijklmnopqrstuvwxyz
==========================
ABCDEFGHIJKLMNOPQRSTUVWXYZ
BCDEFGHIJKLMNOPQRSTUVWXYZA
CDEFGHIJKLMNOPQRSTUVWXYZAB
DEFGHIJKLMNOPQRSTUVWXYZABC
EFGHIJKLMNOPQRSTUVWXYZABCD
FGHIJKLMNOPQRSTUVWXYZABCDE
GHIJKLMNOPQRSTUVWXYZABCDEF
HIJKLMNOPQRSTUVWXYZABCDEFG
...
LMNOPQRSTUVWXYZABCDEFGHIJK
...
```

Pour appliquer le codage de Vigenère, on place le texte à encoder sur la première ligne. Sur la seconde ligne, on place le mot-clef répété. Chaque lettre de la seconde ligne permet de connaître la ligne du tableau de Vigenère à utiliser pour encoder la lettre en clair. Puis on procède la même manière pour toutes les lettres. On constate alors qu'une même lettre du texte clair n'est pas toujours encodé de la même manière.

```
untextesecret
CLEFCLEFCLEFC
WYXJZEIX...
```

Pour coder la première lettre, on commence par regarder la lettre de la seconde ligne ("C") qui désigne l'alphabet encodé à utiliser. Puis on regarde avec cet alphabet quel est la correspondance de la lettre en claire "u", soit "W".

Question a) Finir manuellement l'encodage de l'exemple.

Question b) Construire une fonction qui donne la position d'une lettre dans l'alphabet *Cette fonction sera particulièrement utile dans la suite pour travailler non pas sur un grand tableau de Vigenère, mais en raisonnant sur les indices.*

Question c) En remarquant que la position de la lettre de la clef correspond à la translation (modulo 26) à faire sur la position de la lettre à encoder pour obtenir la lettre codée, proposer une fonction permettant d'encoder ou décoder une lettre à partir de la lettre en clair et la lettre de la clef. ($\star$)

Question d) Écrire une fonction qui encode ou décode un texte à partir d'un mot-clef

5.3 Récursivité

La partie algorithmique intéressante dans les fonctions, se sont les problèmes impliquant la récursivité. Voici quelques exercices plus ou moins difficile dans le genre.

Exercice 45 (Compréhension) *Voici la déclaration d'une fonction récursive :*

```
mccarthy <- function( n ) {
  if ( n>100 ) {
    return(n-10)
  } else {
    return( mccarthy( mccarthy(n + 11) ))
  }
}
```

Question a) Si n est un entier strictement supérieur à 100, quelle est la valeur de `mccarthy(n)` ?

Question b) Et pour $90 \leq nleq100$?

Question c) Et pour $nleq100$ quelconque ?

Exercice 46 (Somme de deux entiers)

Question a) Proposez un algorithme récursif de calcul de la somme de deux entiers naturels a et b . *Vous supposerez que les seules opérations de base dont vous disposez sont :*
- *l'ajout de 1 à un entier a : $a+1$*
- *le retrait de 1 à un entier a : $a-1$*
- *les comparaisons à 0 d'un entier a : $a=0$, $a>0$ et $a<0$.*

Question b) Comment étendre cette fonction aux entiers de signe quelconque ?

Exercice 47 (Qui pair gagne ...) *Écrire les fonctions qui permettent de déterminer récursivement si un nombre est pair ou impair.*

Exercice 48 (Nombre d'or) *Pour $v_1 = 2$, la suite récurrence suivante converge vers le nombre d'or :*

$$v_n = 1 + \frac{1}{v_{n-1}}.$$

Pour rappel, le nombre d'or vaut $\Phi = \frac{1+\sqrt{5}}{2}$. Il est censé être le rapport de distance "le plus esthétique" pour un cadre rectangulaire (en autres rapport de distances) ...
Construire une fonction qui calcule récursivement les valeurs successives de la suite et qui s'arrête lorsque la distance entre deux valeurs successives est inférieure à ϵ ($|v_{n+1} - v_n| < \epsilon$).

Exercice 49 (Suite de Fibonacci) *Possédant au départ un couple de lapins, combien de couples de lapins obtient-on en douze mois si chaque couple engendre tous les mois un nouveau couple à compter du second mois de son existence !*
La suite de Fibonacci répond à cette question :

$$u_0 = u_1 = 1$$
$$\forall n \in \mathbb{N}, u_{n+2} = u_n + u_{n+1}$$

Écrire une fonction récursive en R qui calcule la suite de Fibonacci jusqu'à rang k et qui enregistre les valeurs de $(u_n)_{n\in[0,k]}$ dans un fichier pour être affiché avec un tableur par exemple (format `csv`).

Exercice 50 (Triangle (⋆))

Question a) Récursivité simple sur la hauteur de l'étoile *Écrire une fonction récursive qui affichera un triangle d'étoile d'une hauteur n (cf. illustration ci-dessous). La fonction pourra prendre en paramètre un nombre n, la hauteur totale de l'étoile et m le nombre de niveaux déjà dessinés. De plus, l'utilisation de la récursivité n'interdit pas l'utilisation d'une boucle pour afficher une ligne.*

```
*
**
***
****
*****
```

Question b) Récursivité double(⋆⋆) *Écrire une fonction récursive permettant de dessiner le même triangle mais n'utilisant aucune boucle.*

Exercice 51 (Récursivité sur un tableau (⋆))

Question a) Donnez une version récursive du calcul de la somme des éléments d'un tableau d'entiers.

Question b) Donner une version récursive du calcul du maximum des éléments d'un tableau d'entiers.

Question c) Donnez un algorithme récursif de l'inversion de l'ordre des éléments d'un tableau.

Exercice 52 (Recherche Dichotomique (⋆))

Question a) Donner une version récursive d'un algorithme de recherche d'un élément dans un tableau d'entiers.

Question b) Faire de même en supposant que le tableau est ordonné

Exercice 53 (PGCD)

Question a) En utilisant l'algorithme d'Euclide, écrire une fonction **pgcd** qui calcule le pgcd de deux entiers naturels (en utilisant **while**) a et b. ? *Rappel : PGCD= Plus Grand Commun Diviseur. Consulter Wikipedia pour trouver un exemple d'algorithme d'Euclide.*

Question b) En notant que le pgcd de a et b peut être obtenu comme le pgcd de b et $\frac{a}{b}$ (division entière) si b n'est pas nul alors, écrire une fonction **pgcdr** qui calcule également le pgcd de a et b en utilisant une stratégie récursive

Question c) En utilisant de grands nombres, comparer les performances des deux méthodes.
Aide : pour obtenir le temps de calcul de l'exécution d'une fonction **toto***(), vous pouvez procéder de la sorte :*

```
ptm <- proc.time() # ptm retient l'heure de debut d'execution
toto()
print( proc.time() - ptm ) # on affiche la difference avec l'heure de fin
```

Les deux problèmes suivants peuvent se traiter de manière récursive ou iterative plus ou moins simplement. Ils ont également été choisit pour illustrer des problèmes de décidabilité (que nous n'aborderons pas ...).

Exercice 54 (Lychrel numbers (⋆⋆)) *Si on prend 47, qu'on le retourne et qu'on l'ajoute à lui-même ($47 + 74 = 121$), on obtient un palindrome.*
Cela n'arrive pas nécessairement à tous les coups, mais si on repète cette opération, alors on peut retomber sur des palindromes ... par exemple :

$$349 + 943 = 1292$$
$$1292 + 2921 = 4213$$
$$4213 + 3124 = 7337$$

Pour 349*, il faut donc 3 étapes pour arriver à un palindrome.*
Actuellement, personne n'a prouvé qu'il était possible de trouver un palindrome pour tout le nombre entier (problème de décidabilité). Les mathématiciens pensent même que certains ne permettent pas de produire de palindrome. Par exemple, en répétant un très grand nombre de fois l'opération 196 *ne produit pas de palindrome ... mais peut être faut il répéter l'opération encore plus de temps ... De tels nombres sont appelés des Nombres de Lychrel.*
Pour les raisons du problème, nous dirons qu'un nombre est pseudo-Lychrel si il n'est pas possible de trouver un trouver un palindrome avant 50 iterations de la transformation.

Question a) Implémenter une fonction qui retourne un nombre ? ($\star\star$)

Question b) Implémenter une fonction qui retourne 1 si le nombre est un palindrome et 0 sinon ($\star$)

Question c) Implémenter une fonction itérative permettant de déterminer si un nombre est pseudo-Lychrel

Question d) Implémenter une fonction récursive permettant de déterminer si un nombre est pseudo-Lychrel ($\star$)

Question e) Quel est le premier nombre pseudo-Lychrel qui n'est pas Lychrel ?

Question f) Combien y a-t-il de nombres de pseudo-Lychrel avant 10000 ?

Exercice 55 (Collatz Problem ($\star$)) *Soit n un entier positif, et considérons la transformation suivante :*
- $n \to n/2$ *si n est pair*
- $n \to 3n+1$ *sinon*

En appliquant récursivement ces règles sur le chiffre 13, on obtient la séquence suivante :

$$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

On peut voir que cette séquence finit à 1 et contient 10 termes. Actuellement, il n'a pas été prouvé que pour tout nombre entier, la séquence finit bien à 1 (conjecture de Collatz, c'est de nouveau un problème de décidabilité).

Question a) Écrire un programme récursif qui calcule la longueur du chemin jusqu'à 1 pour un nombre entier n ($\star$) *Aide : on utilisera une fonction avec le profil ci-dessous :*

```
# n : nombre entier sur lequel appliquer la regle jusqu'a 1
# retourne la taille du chemin pour aller de n a 1
CollatzRec <- function(n)
```

Question b) Écrire un programme itératif qui calcule la longueur du chemin jusqu'à 1 pour un nombre entier n ($\star$)

Question c) Quel chiffre inférieur à 100000 produit la plus longue chaîne ?

CHAPITRE V

Solutions aux exercices

Solution à l'exercice 1 Dans l'ordre des questions :
- Faire un court tableau de variables pour voir que x=19 et y=19.
- Faire un court tableau de variables pour voir que x=12 et y=12.
- Faire un court tableau de variables pour voir que x=31, y=31 et z=31.
- Faire un court tableau de variables pour voir que x=12, y=12, y=12 et z=12.

```
A <- A+B
B <- A-B
A <- A-B
```

ou classiquement

```
C<-A
A<-B
B<-C
```

Solution à l'exercice 2 Extension du cas à deux variables :

```
t <- x
x <- y
y <- z
z <- t
```

Solution à l'exercice 3
- Le programme de gauche affiche respectivement :
  - `i -1 est fitagen`
  - `i 2 est fitisop`
  - `i 4 est fitisop`
  - `i 6 est fitisop`
- Le programme de droite affiche respectivement :
  - `ici?`
  - `ici?`
  - `ici?`
  - `la?`

Solution à l'exercice 4

Question a) *En imaginant que les variables ont été initialisées comme ci-dessous, indiquer ce qu'affichera le programme* Faire un tableau d'évolution de variable pour voir que le résultat sera

```
x=22.3
y=23.4
z=120
```

Question b) *Compléter les trois lignes de commentaires du programme ci-dessous.* Une aide est indiqué par le fait de s'intéresser à l'ordre des variables dans les assertions.
Pour le premier commentaire, on peut écrire que `ICI x<=y` (1), en effet, si `x<=y` à la ligne 3, le test et faux et rien de change. Si `x>y` alors on inverse les variables (*cf.* exercice), donc après l'opération, on a `y>x`. Notez que `w` n'existe pas nécessairement !
Pour le second commentaire, on peut écrire que `ICI z>=y et z>=x` (2). On a directement que `z>=y` de même que précédemment. Ensuite, si `z>=y` à la ligne 10, alors on ne change rien et comme `x<=y` (1), on en déduit le commentaire. Si `y>=z`, alors on inverse `y` et `z`, et donc ce qui était vrai pour `y` avant est vrai pour `z` maintenant. On a donc, d'après la relation (1), `x<=z`.
Pour le troisième commentaire, on peut écrire que `ICI z>=y et y>=x` (3). On a directement que `y>=x` de même que précédemment. Si `y>=x` à la ligne 14, alors rien ne change et on en déduit facilement (3) car `z>=y` d'après

(2). Dans le cas contraire, on inverse `y` et `x`, mais du coup comme on avait `z>=x`, on a maintenant `z>=y`, on en déduit donc (3).

Question c) *Conclure sur ce que fait le programme ?* Le programme ordonne les trois valeurs par ordre croissant.

Solution à l'exercice 5 Exemple de tableau avec une saisie de la valeur 2.

line	a	b	i	prod	i<b	b>0
1	3	?	?	0		
2	3	?	?	0		
3	3	2 (!)	?	0		
4	3	2	?	0		true
5	3	2	0	0		
6	3	2	0	0	true	
7	3	2	0	3		
8	3	2	1	3		
6	3	2	1	3	true	
7	3	2	1	6		
8	3	2	2	6		
6	3	2	2	6	false	
10	3	2	2	6		

Le programme affiche la multiplication de `a` par `b` si `b` est positif ou nul, et affiche 0 sinon.

Solution à l'exercice 6

Question a) *Compléter les commentaires*
Dans la boucle ;
- $j = j0 - n + 1$ en début de boucle, et en considérant que n vaut 1 lors du premier passage.
- $d = \frac{d0}{2^{n-1}}$ car à chaque tour de boucle on divise la valeur initiale par 2. Avec la relation précédente, on en déduit que $d = \frac{d0}{/} 2^{j0-j}$.

À la sortie de la boucle, on sait que j vaut 1, on en déduit donc que $d = \frac{d0}{/} 2^{j0-1}$.

Question b) *En déduire ce que fait le programme ?* Le programme affiche le résultat du calcul suivant $d = \frac{d0}{/} 2^{j0-1}$

Solution à l'exercice 7

Question a) *Exprimer la valeur de* `f` *dans la boucle en fonction de* $f0 = 81$ *et de* **nbTours**. On note $f0 = 81$, la valeur initiale de f. À chaque tour de boucle, on a `f <-sqrt(t)`, et **nbTours** indique le nombre de tour, on en déduit donc que le commentaire permet de préciser l'invariant de boucle suivant :

$$f = \sqrt[nbTours]{f0},$$

en supposant que $\forall x,\ \sqrt[0]{x} = x$

Question b) *En déduire, l'affichage du programme* On peut répondre d'abord intuitivement à la question. À chaque tour de boucle, on applique une racine carrée, la valeur de f est donc successivement 81, 9 et 3. C'est donc au troisième tour que la valeur de f est inférieur à 5. Et donc le programme affiche 3.
Maintenant, on peut raisonner analytiquement, ce qui permettrait d'avoir une réponse quelque soit la valeur initiale de f. À la fin de la boucle on a donc $f > 5$, soit d'après la relation précédente $\sqrt[nbTours]{f0} > 5$ et donc $f0^{\frac{1}{2^{nbTours}}} > 5$. La fonction log étant strictement croissante, on en déduit que $\frac{1}{2^{nbTours}} * \log f0 > \log 5$ ... on en déduirait alors analytiquement une valeur limite de n sous la forme d'un $\log(\log(\dots))$.

Question c) *Mêmes questions en remplaçant la condition du* `while` *par* `f>0` Théoriquement on sait que $\forall x,\ \sqrt{}(x) > 0$, comme f est calculé comme une racine carrée, alors le test `f>0` ne sera jamais faux. Le programme fait donc une boucle infinie et n'affiche jamais rien.
Néanmoins, vous pourrez observer en pratique que ce programme fini par s'arrêter à cause des approximations des nombres dans un ordinateur.

Solution à l'exercice 8

```
i <- 100
while( i >=0 ) {
  cat(i, "\n")
  i <- i-1
}
```

Solution à l'exercice 9

```
nombre <- 7
cat("Table de ", nombre, "\n")
i <- 1
while( i<10 ) {
  val <- nombre * i
  cat(nombre, " x ", i, " = ", val, "\n")
  i <- i+1
}
```

Solution à l'exercice 10 Il s'agit d'un algorithme utilisant un accumulateur qui va retenir progressivement les valeurs de la forme $fact = 1 \times 2 \times 3 \times \cdots \times i$. Ceci permet de donner un invariant de boucle. $fact$ est initialisé avec la valeur initiale de la "suite", soit 1.

```
val <- 8
fact <- 1
i <- 1
while( i<=val ) {
  #ICI fact=1 x 2 x 3 x  ...  x i
  i <- i+1
  fact <- fact * i
}
#ICI fact=1 x 2 x 3 x  ...  x val
```

Solution à l'exercice 11 Si on arrive à la ligne 6, on peut écrire que `i` est impair. Car dans le cas contraire, on serait allé à la ligne 4 et on aurait "sauté" directement à la ligne 2.
Si on arrive à la ligne 9, on peut écrire que `i` est impair et que `i` est un multiple de 3. On passage, on peut remarquer qu'il s'agit uniquement des valeurs 3, 9, 15, 21, 27, ..., 45. (valeurs de 6 en 6).
Si on arrive en ligne 12, on n'est pas allé en ligne 9, sinon le break aurait conduit à "sauter" directement en ligne 14 (fin de boucle). Donc `i` est impair et `i` n'est pas un multiple de 3, soit les valeurs 1, 5, 7, 11, 13, 17 ... il serait plus difficile de donner une relation générale pour cette suite.

Solution à l'exercice 12 Cet exercice permet de manipuler des accumulateurs.

Question a) *Calculer des sommes suivantes*
Somme des n premiers carrés : $\sum_{i=1}^{n} i^2$

```
S<-0
for (i in 1:n) {
  S <- S+i*i
  #ICI P vaut 1*1+2*2+3*3...+i*i
}
```

Somme des n premiers entiers pairs (plusieurs solutions possibles),

```
S<-0
for (i in 1:n) {
  if( i%2==0 ) {
    S <- S+i
  }
}
```

ou encore

```
S<-0
i<-0
while (i<n/2) {
  S <- S+i
  i<-i+2
}
```

Somme des n premiers entiers impairs (plusieurs solutions possibles).

```
S<-0
for (i in 1:n) {
  if( i%2==1 ) {
    S <- S+i
  }
}
```

ou encore

```
S<-0
i<-1
while (i<n/2) {
  S <- S+i
  i<-i+2
}
```

Question b) *Transformer les boucles* `while` *en* `for`. à faire ...

Question c) *Produit des n premiers entiers.*
De manière similaire au cours, on peut écrire $P = \prod_{i=1} ni$, et en déduire un programme très similaire :

```
P<-1
for (i in 1:n) {
  P <- P*i
  #ICI P vaut 1*2*3*...*i
}
```

Attention à l'initialisation !

Solution à l'exercice 13

Question a) *Valeur approchée de* π
Il s'agit du calcule d'une somme avec un accumulateur dont le terme général est $\frac{1}{i^2}$.

```
p<-20000
S<-0
for (i in 1:p) {
  S <- S + 1/(i*i)
}
pit <- sqrt(S*6)
cat("valeur approchee de pi : ", pit, "\n")
```

NB : la valeur de p est plutôt grande car la méthode est dite à "convergence lente", il faut beaucoup d'itération pour arriver à déterminer des décimales de π (ici, on a 4 décimales exactes).

Question b) *Afficher à chaque itération la valeur de l'accumulateur*
Dans ce programme, on est obligé d'inclure le calcul de la valeur approchée dans la boucle. En faisant de la sorte, on va être obligé de faire faire beaucoup de calculs de racine carrée ... ce qui est long pour l'ordinateur (je réduis donc p).

```
p<-2000
S<-0
for (i in 1:p) {
  S <- S + 1/(i*i)
  pit <- sqrt(S*6)
  cat(pit, "\n")
}
cat("valeur approchee de pi : ", pit, "\n")
```

Question c) *Afficher la différence avec la valeur "réelle" de* π

```
p <- 20000
S <- 0
for (i in 1:p) {
  S <- S + 1/(i*i)
  diff <- abs(pi - sqrt(S*6))
  cat(diff, "\n")
}
cat("valeur approchee de pi : ", sqrt(S*6), "\n")
```

Question d) *Modifier le programme pour qu'il calcule tant que la différence à la valeur réelle est supérieure à une précision*
Dans ce programme, i sert de compteur de boucle pour répondre à la question, mais on ne s'en sert plus dans le test d'arrêt de la boucle.

```
p <- 10^-6
S <- 0
i <- 1 # on commence a 1 pour eviter un bug de division par 0
diff <- 1 # on met une valeur aleatoire mais superieure p pour rentrer dans la boucle
```

```
while(  diff >p ) {
  S <- S + 1/(i*i)
   diff  <- abs(pi - sqrt(S*6))
  i  <- i+1
}
cat("valeur approchee de pi : ", sqrt(S*6), ", diff=", diff, " atteint au bout de ", i, " tours de boucles\n")
```

En faisant faire les calculs, on obtient 6 chiffres de précision pour 954931 tours de boucle. La convergence est donc TRÈS lente ...

Solution à l'exercice 14

Question a) *Tableau avec les positions d'indice pair et de 1 sinon.* Pour une raison d'efficacité, il est mieux de contruire un tableau `m` à la bonne taille au départ, puis de le remplir que d'utiliser la propriétés d'auto-agrandissement.

```
n <-15
m <- rep(0, n) #construit un vecteur de n 0
for(i in 1:n) {
  if( i%2==0 ) {
    #ICI i est pairs
    m[i] <- 1
  } else {
    m[i] <- 0
  }
}
```

Notez que le `else` est inutile !

Question b) *Tableau décalé*

```
n <-15
val <- 2
m <- rep(0, n)
for(i in 1:n) {
  m[i] <- val
  val <- val + 1
}
```

Question c) *Tableau inversé* D'abord, i sert d'itérateur des cases du tableau :

```
n <-15
m <- rep(0, n)
for(i in 1:n) {
  val <- -n+i
  m[i] <- val
}
```

autre solution (i sert d'iterateur des valeurs successives)

```
n <-15
m <- rep(0, n)
i <- 1
while(i<=n) {
  m[n-i] <- -i
  i <- i+1
}
```

Solution à l'exercice 15

Question a) *Somme des éléments d'un tableau* `T`

```
S <- 0
i <- 1
while(i<=length(T)) {
  S <- S + T[i]
  i <- i+1
}
```

Question b) *Moyenne des éléments d'un tableau* `T` On le programme précédent et on ajoute la ligne suivante à la fin :

```
moy <- S/length(T)
```

Question c) *Variance des éléments d'un tableau* `T` On commence par calculer la moyenne comme avant, puis on effectue l'algorithme suivant :

```
V <- 0
i <- 1
while(i<=length(T)) {
  V <- V + (moy-T[i])**2
  i <- i+1
}
V <- V/length(T)
```

Solution à l'exercice 16

```
i <- 1
while(i<=length(T)) {
  if( T[i]==23 ) {
    T[i]==32
  }
  i <- i+1
}
```

Solution à l'exercice 17 L'algorithme ci-dessous permet vérifier case par case (iteration avec i) si il y a des doublons. Les case contenant des 0 ne sont pas traitées. Pour une case ne contenant pas un 0, on regarde pour toutes les cases du tableau (itération avec j) si on a un doublon, c'est-à-dire si `T[i]==T[j]`. Si tel est le cas, alors la case `T[j]` doit être mise à 0 et on doit poursuivre pour regarder les autres doublons de `T[i]`. On utilise un *flag* pour se souvenir que `T[i]` est un doublon et qu'il faudra le mettre à zéro plus tard.

```
i <- 1
while(i<=length(T)) {
  # ICI, il n'y a pas de doublons autres que des 0 pour le sous-tableau [1,i]
  if( T[i]==0 ) {
    next
  }
  j <- 1
  doublon <- F
  while(j<=length(T)) {
    if( i==j ) {
      #on ne doit pas traiter cette case
      next
    }
    if( T[i]==T[j] ) {
      doublon <- T
      T[j] <- 0
    }
    j <- j+1
  }
  if( doublon ) {
    #ICI T[i] avait un doublon
    T[i] <- 0
  }
  #on passe a la case suivante
  i <- i+1
}
```

Solution à l'exercice 18

```
i <- 1
S <- 0
while(i<=length(T1)) {
  j <- 1
  while(j<=length(T2)) {
    S <- S + T1[i]*T2[j]
    j<-j+1
  }
  i <- i+1
}
print(S)
```

Solution à l'exercice 20 On fait une recherche en partant de la fin, et on s'arrête dès qu'on trouve un élément.

```
i <- length(tab)
po< <- -1
while( i>=1 ) {
  if ( elem==tab[i] ) {
    pos <- i
    break
  }
  i <- i-1
}
#ICI, si pos == -1 alors l'element n'a pas ete trouve
# sinon, pos donne la position du dernier element elem
```

On pourrait très bien faire une recherche depuis le début sans `break`, mais avec une efficacité moyenne inférieure :

```
pos <- -1
for( i in 1:length(tab) ) {
  if ( elem==tab[i] ) {
    pos <- i
  }
}
#ICI, si pos == -1 alors l'element n'a pas ete trouve
# sinon, pos donne la position du dernier element elem
```

Solution à l'exercice 21

```
nb <- 0
for( i in 1:length(tab) ) {
  if ( val==T[i] ) {
    nb <- nb+1
  }
}
```

Solution à l'exercice 24 On utilise la propriété d'auto-accroissement des tableaux

```
#ICI Tab est un tableau de nombres et e est un nombre
Tout <- c() #vecteur vide
j <- 1 # j est la position suivante a remplir dans le tableau de sortie Tout
for( i in 1:length(Tab) ) {
  if ( Tab[i]==e ) {
    #ICI, on ne traite pas cet element et on passe a la case suivante
    next
  }
  Tout[j] <- Tab[i]
  j <- j+1
}
```

Autre approche :

```
#ICI Tab est un tableau de nombres et e est un nombre
Tout <- c() #vecteur vide
for( i in 1:length(Tab) ) {
  if ( Tab[i]==e ) {
    next
  }
  #ICI Tout
  Tout[ length(Tout)+1 ] <- Tab[i]
}
```

Solution à l'exercice 25 Première solution simple :

- pour une cellule à la position `i`, on regarde si la valeur est la même que pour la cellule "symétrique", à la position `l - (i-1)`, on en déduit le test,
- dès qu'on constate une disymétrie, le tableau n'est pas un palindrome, donc on arrête le parcours avec un `break`,
- on utilise une variable `ispalindrome` comme variable retenant l'information qu'on a observé une disysmétrie.

```
ispalindrome <- T
l <- length(tab)
for( i in 1:l ) {
  if ( tab[l - (i-1)]!=tab[i] ) {
    ispalindrome <- F
    break
  }
}
```

Dans cette première solution, on teste deux fois la symétrie en faisant varier l'indice de 1 à `l`. On propose donc une seconde solution s'arrêtant lorsque la moitié du tableau a été parcourus, c'est suffisant.
Dans la mesure où `l` est pair ou impair, on utilise l'opérateur de division entière `%/%` pour savoir quant s'arrêter. On a `4%/%2` qui vaut 2 mais également `5%/%2` qui vaut 2. Vous pourrez vérifier que cet algorithme fonctionne pour les cas de taille de tableau pair et impair.

```
ispalindrome <- T
l <- length(tab)
for( i in 1:l%/%2 ) {
  if( tab[l - (i-1)]!=tab[i] ) {
    ispalindrome <- F
    break
  }
}
```

D'autres solutions sont possibles. La solution ci-dessous utilise deux indice `i` va du début vers le milieu et `j` va de la fin au début.

```
ispalindrome <- T
l <- length(tab)
i <- 1
j <- l
while( i <= l%/%2 && j>= l%/%2 ) {
  if( tab[i]!=tab[i] ) {
    ispalindrome <- F
    break
  }
  i <- i+1
  j <- j-1
}
```

encore une autre solution, un peu plus astucieuse, sans `break` mais qui s'arrête lorsqu'on a vu que ce n'était pas un palindrome. Le `if` est requis dans cette solution pour le cas d'un

```
ispalindrome <- T
l <- length(tab)
i <- 1
while( i<=l%/%2 && tab[l-(i-1)]!=tab[i] ) {
  i <- i+1
}

if( i<l%/%2 || tab[l-(l%/%2-1)]!=tab[l%/%2] ) {
  ispalindrome <- F
}
```

Solution à l'exercice 26

Question a) *Algorithme avec tableau intermédiaire*

```
T2 <- rep(0, length(tab)) # creation d'un tableau de la meme taille que T2 remplis de 0
l <- length(tab)
for( i in 1:l ) {
  T2[l - (i-1)] <- tab[i]
}
```

autre solution avec un second indice `j` et une boucle `while` :

```
T2 <- rep(0, length(tab)) # creation d'un tableau de la meme taille que T2 remplis de 0
l <- length(tab)
i <- 1
j <- l
while( i <=l ) {
  T2[j] <- tab[i]
  j <- j-1
  i <- i+1
}
```

Question b) *Version "en place"* L'idée de cet algorithme est d'inverser le contenu de la cellule `tab[l - (i-1)]` avec celui de la cellule `tab[i]`. Bien évidemment, il ne faut traiter que la moitié du tableau pour ne pas inverser deux fois !

```
l <- length(tab)
for( i in 1:l%/%2 ) {
  # inversion des variables tab[l - (i-1)] et tab[i]
  val <- tab[l - (i-1)]
  tab[l - (i-1)] <- tab[i]
  tab[i] <- val
}
```

Solution à l'exercice 27

Question a) *Algorithme avec tableau intermédiaire*
– Attentions aux bords ... on commence à 2
– On traite le cas de `tab[1]` à la fin

```
l <- length(tab)
T2 <- rep(0, l) # creation d'un tableau de la meme taille que T2 remplis de 0

#on decale vers la droite
for( i in 1:l-1 ) {
  T2[i+1] <- tab[i]
}

#le premier vaut le dernier
T2[1]<-tab[l]
```

autre solution avec un second indice `j` et une boucle `while` :

```
l <- length(tab)
T2 <- rep(0, l) # creation d'un tableau de la meme taille que T2 remplis de 0
i <- 1
j <- 2
while( i<l ) {
  T2[j] <- tab[i]
  j <- j+1
  i <- i+1
}

#le premier vaut le dernier
T2[1]<-tab[l]
```

Question b) *Version "en place"* La difficulté de la version en place, c'est la perte d'une information dans une cellule lorsqu'on écrit dessus. Il est donc nécessaire de faire attention à parcourir le tableau dans le bon sens et à traiter de manière spécifique le cas de la première modification.
On va commencer par traiter le cas du déplacement vers la gauche :

```
l <- length(tab)
#on decale vers la gauche
for( i in 2:l ) {
  tab[i-1] <- tab[i]
}
#le dernier vaut l'ancien premier
tab[l] <- val
```

Si on souhaite décaler vers la droite, il faut faire attention au sens de parcours, le programme devient :

```
l <- length(tab)
i <- l-1
while( i>=1 ) {
  tab[i+1] <- tab[i]
  i <- i-1
}
#le dernier vaut l'ancien premier
tab[l] <- val
```

Solution à l'exercice 28

```
l <- length(tab)
cl <- 1 # longueur de la sequence en cours
ml <- 1 # longueur max (necessairement >=1)
cv <- tab[1] # valeur de la sequence en cours
#on decale vers la gauche
```

```
for( i in 2:l ) {
  if( tab[i]==cv ) {
    # ICI la valeur courante est la meme que celle de la sequence en cours
    # on incremente cl
    cl <- cl+1

    #on met a jour la valeur maximale ml
    if( cl>ml ) {
      ml <- cl
    }
  } else {
    #ICI la valeur courante met fin a la sequence en cours et en commence donc une nouvelle
    cl <- 1
    cv <- tab[i]
  }
}
```

Solution à l'exercice 29

```
v <- M[1,] #le vecteur c contient la premiere ligne de M
for( i in 1:nrow(M) ) {
  # i est l'indice de ligne
  for( j in 1:ncol(M) ) {
    # j est l'indice de colonne
    if( v[j]>M[1,j] ) {
      v[j] = M[1,j]
    }
  }
}
print(v)
```

Solution à l'exercice 30

Question a) *Algorithme en deux étapes*

```
#calcul du max de T
m <- T[1]
for( i in 1:length(T) ) {
  if( T[i] > m ) {
    m <- T[i]
  }
}

#recherche du max m dans le tableau T
idx <- c()
nb <- 1
for( i in 1:length(T) ) {
  if( T[i] == m ) {
    idx[nb] <- i
    nb <- nb+1
  }
}
```

Question b) *Parcourt unique*

```
#calcul du max de T
m <- T[1]
idx <- c(1) # creation d'un vecteur de taille 1 contenant la valeur 1
for( i in 2:length(T) ) {
  # ICI idx donne la liste des indices <i de l'element maximal
  #

  if( T[i] > m ) {
    m <- T[i]
    idx <- c(i) # creation d'un vecteur de taille 1 contenant la valeur i
    nb <- 1
  } else if ( T[i]==m ) {
    idx[nb] <- i
    nb <- nb+1
  }
}
```

Solution à l'exercice 31

Question a) *Conversion des Euros en Dollars, conversion de des Dollars en Yen.*

```
DtoY <- function(vald) {
  valy <- 91.7*vald
  return(valy)
}
YtoD <- function(valy) {
  vald <- valy/91.7
  return(vald)
}
EtoD <- function(vale) {
  vald <- vale/0.73
  return(vald)
}
DtoE <- function(vald) {
  vale <- 0.73*vald
  return(vale)
}
```

Question b) *Conversion des Euros en Yen*

```
EtoY <- function(vale) {
  vald <- EtoD(vale)
  valy <- DtoY(vald)
  return(valy)
}
YtoE <- function(valy) {
  vald <- YtoD(valy)
  vale <- DtoE(vald)
  return(vale)
}
```

Question c) *Écrire une fonction qui calcule l'aire d'un rectangle à partir de sa longueur et de sa largeur*

```
aire <- function(l, L) {
   a <- l*L
   return(a)
}
```

Solution à l'exercice 32

Question a) *Construire une fonction* **ricker**

```
ricker <- function(nzero, r, K=1, time=100, from=0, to=time)
{
  N <- numeric(time+1)
  N[1] <- nzero
  for (i in 1:time) {
    N[i+1] <- N[i]*exp(r*(1 - N[i]/K))
  }
}
```

Question b) *Utiliser votre fonction pour dessiner l'évolution de la pollution en fonction des générations.*

```
Time <- 0:time
N <- ricker(0.1,2)
plot(Time, N, type="l", xlim=c(from, to))
```

Question c) *Écrire une fonction* **ricker.draw** *qui* ***fait la même chose*** *que* **ricker** *avec pour effet de bord de dessiner l'évolution de la population.*

```
ricker.draw <- function(nzero, r, K=1, time=100, from=0, to=time)
{
  N <- numeric(time+1)
  N[1] <- nzero
  for (i in 1:time) {
    N[i+1] <- N[i]*exp(r*(1 - N[i]/K))
  }
  Time <- 0:time
  plot(Time, N, type="l", xlim=c(from, to))
}
```

Solution à l'exercice 33

Question a) *Fonction* **estpremier**

```
estpremier <-function(n)
{
  for(i in 2:(n-1)) {
    if( i%%n == 0 ) {
      return(FALSE)
  }
  return(TRUE)
}
```

Question b) *Premiers entiers inférieurs à p*

```
p<-1000
premiers <- c()
for( e in 1:p) {
  if( estpremier(e) ) {
    premiers <- c(premiers, e) #ajoute e a la  liste  premiers
  }
}
```

Question c) *Fonction* **estpremier**

```
estpremier <-function(n)
{
  for(i in 2:(n%/%2) ) {
    if( i%%n == 0 ) {
      return(FALSE)
  }
  return(TRUE)
}
```

Question d) *Premiers entiers inférieurs à p (2)* Idem avant, simplement, cela prend 2 fois moins de temps !

Question e) *Algorithme d'Erathostenes*

```
Eratosthenes <-function(n) {
  if (n >= 2) {
    sieve <- seq(2, n)
    primes <- c()
    for (i in seq(2, n)) {
      if (any(sieve == i)) {
        primes <- c(primes, i)
        sieve <- c(sieve[(sieve %% i)~!= 0], i)
      }
    }
    return(primes)
  } else {
    stop("Input value of n should be at least 2.")
  }
}
```

Autre solution

```
cribleErathostene <- function(n) {
    x <- 1:n
    positionsBarres <- rep(FALSE,n) #vecteurs de valeurs booleennes fausses
    for (i in 2:length(x)) {
        if ( (x[i]*2 < n) && (!positionsBarres[i]) ){
            for (j in seq(2*x[i],n,by=x[i]) ) {
                positionsBarres[j] <- TRUE
            }
        }
    }
    return(x[!positionsBarres])
}
cribleErathostene(100)
```

Solution à l'exercice 35

Question a) *Somme pour des entiers naturels*

```
somme <- function(a,b) {
  if( a>0 ) {
    val <- somme(a-1,b)+1
    return(val)
  } else if( b>0 ) {
      val <- somme(a,b-1)+1
      return(val)
  } else {
    return(0)
  }
}
```

Question b) *Somme pour des entiers relatifs*

```
somme <- function(a,b) {
  if( a>0 ) {
    val <- somme(a-1,b)+1
    return(val)
  } else if( a<0 ) {
    val <- somme(a+1,b)-1
    return(val)
  } else {
    #ici a==0
    if( b>0 ) {
      val <- somme(a,b-1)+1
      return(val)
    } else if( b<0 ) {
      val <- somme(a,b+1)-1
      return(val)
    } else {
      return(0)
    }
  }
}
```

Solution à l'exercice 36 Il s'agit d'une récursivité mutuelle. On implémente deux fonctions récursives.

```
impair <- function(n) {
  if( n==0) {
    return(FALSE)
  } else if(n==1) {
    return(TRUE)
  } else {
    return(pair(n-1))
  }
}

pair <- function(n) {
  if( n==0 ) {
    return(TRUE)
  } else if(n==1) {
    return(FALSE)
  } else {
    return(impair(n-1))
  }
}
```

Solution à l'exercice 37

Question a) *Fonction récursive*
Voici la fonction récursive avec des n décroissants qui permet de calculer $v(n)$, mais ce n'est pas ce qu'on veut faire ici.

```
v<-function(n) {
  if( n==1 ) {
    return(2)
  } else {
    return( 1+1/v(n-1)
  }
}
```

Pour répondre à la question, il faut une fonction qui fait une récursion sur le raffinement progressif du calcul.

```
recur<-function(vpred, epsilon) {
  vnext <- 1+1/vpred
  print(vnext)
  if( abs(vnext-vpred) > epsilon ) {
    recur(vnext, epsilon)
  }
}

eps <- 0.001
recur(2, eps)
```

Question b) *(⋆) Construire une fonction qui calcule récursivement les valeurs successives de la suite et qui s'arrête au bout de p appels récursifs.* La difficulté de cette question réside dans le faite de trouver un moyen de savoir quel est nombre d'appel récursif qui ont été fait.

```
recur<-function(vpred, p, level=0) {
  vnext <- 1+1/vpred
  print(vnext)
  if( level < p ) {
    recur(vnext, p, level+1)
  }
}

nb <- 20
recur(2, nb, 0)
```

Question c) *Version itérative*

```
nb <- 20
v<-2
for(i in 1:nb) {
  v <- 1+1/v
  print(v)
}
```

Solution à l'exercice 38 Il faut décaler la relation de récurrence!!
Version sans tableau :

```
Fibo <- function() {
    if(n==0 || n==1) {
      return(1)
    } else {
      val <- Fibo(n-2) + Fibo(n-1)
      return(val)
    }
}
```

Les techniques usuelles pour traiter les cas des tableaux ne fonctionnent pas en R :
- les tableaux ne peuvent être passés en paramètres pour être modifiés car on ne récupère pas la modification du tableau,
- l'utilisation d'effet de bord ne fonctionne pas ...

Du coup, la solution est un peu étrange ...

```
Fibo <- function(n) {
    if(n==1) {
      T[1] <- 1
      T[2] <- 1
      return(T)
    } else {
      T<-Fibo(n-1)
      val <- T[n] + T[n-1]
      T[n+1] <- val
      return(T)
    }
}

Tab <- Fibo(15)
```

Solution à l'exercice 39

Question a) *recursivité simple*

```
pyramide <- function(n) {
  if( n!=0 ) {
    for(i in 1:n) cat("*")
    cat('\n')
    pyramide(n-1)
  }
}
```

Question b) *double recursivité*

```
pyramide <- function(n, p) {
  if( n!=0 ) {
    if(n<p) {
      cat('\n')
      pyramide(n-1, 1)
    } else {
      cat('*')
      pyramide(n, p+1)
    }
  }
}

pyramide(5, 1)
```

Solution à l'exercice 42

Question a) *Algorithme d'Euclide*

```
pgcd <- function (a,b) {
    while ( a%%b != 0) {
        reste <- a%%b
        a <- b
        b <- reste
    }
    return(b)
}

pgcd(50,30)
```

Question b) *Version recursive*

```
pgcdr <- function (a,b) {
  if (b == 0) return(a)
  return(pgcdr(b,a%%b))
}

pgcdr(50,30)
```