

★ **Exercice 1:** Code mystère.

- ▷ **Question 1:** Calculez les valeurs renvoyées par la fonction `f` pour `n` variant entre 1 et 5.
- ▷ **Question 2:** Quelle est la fonction mathématique vue en cours que `f()` calcule ?
- ▷ **Question 3:** Quelle est la complexité algorithmique du calcul ?

```
def f(n:Int):Int = {
  def lambda(n:Int, a:Int, b:Int):Int = {
    if (n == 0) {
      return a;
    } else {
      return lambda(n-1, b, a+b);
    }
  }
  return lambda(n, 0, 1);
}
```

Notez que tout le travail est fait par la fonction interne `lambda`, et la fonction `f` ne sert qu'à donner une valeur initiale aux arguments `a` et `b`, qui servent d'accumulateur. Il s'agit là d'une technique assez classique en récursivité.

Réponse

C'est fibonacci, bien sûr. Et c'est bô car c'est du récursif avec une complexité algorithmique linéaire. C'est l'occasion de dire que si la forme classique de fibo est aussi nulle en perf, c'est pas tant à cause de la récursivité, mais plutôt à cause de la façon dont elle est écrite, c'est tout.

Remarquez aussi que les algos classiques itératifs sont linéaires en temps, mais également linéaires en espace vu qu'ils font un gros tableau des résultats temporaires déjà rencontrés. Ce n'est pas le cas de cette approche, qui est bien sûr aussi applicable en itératif.

Le message est "la récursion n'est ni plus ni moins efficace que l'itératif" et "Cette approche est la plus efficace que je connaisse" (même si je m'amuserais pas à avancer qu'elle est optimale, il est possible qu'une fonction en temps constant existe pour calculer fibo, après tout).

Il faut également noter qu'en Scala, on a tout à fait le droit de définir une fonction à l'intérieur d'une autre fonction. Cela limite sa visibilité comme il se doit, ce qui est pratique vu que cette `lambda` n'a aucun sens hors contexte.

Fin réponse

★ **Exercice 2:** Soit le type `List[Char]` muni des opérations suivantes :

<code>Nil</code>	La liste vide
<code>head :: tail</code>	Construit une liste constituée de <code>head</code> , suivi de la liste <code>tail</code>
<code>list.head</code>	Récupère le premier caractère de la liste (pas défini si <code>list</code> est la chaîne vide)
<code>list.tail</code>	Récupère la liste privée de son premier élément (idem)

Écrire les fonctions suivantes. Vous préciserez les préconditions nécessaires.

Réponse

Comment lancer la séance : "les ptits malins qui savent déjà tout, vous avez 20 questions devant vous, pas la peine de nous attendre, on va prendre le temps de comprendre. Avancez. Indication : toutes les questions admettent des réponses linéaires en temps, sauf 2. L'une est un peu pire que $O(n)$, l'autre est meilleure. Bonne chance. En cas de doute, vous pourrez taper ce code dans la PLM d'ici peu, car on est en train d'ajouter leçon sur les listes récursives". Et ensuite, on prend les vrais débutants par la main.

À propos des preuves On aborde la notion de preuve plus tard dans le cours, et il s'agit donc de faire un peu avec les mains dans ce TD, pour préparer le terrain au cours. Il faut absolument évoquer la preuve de terminaison de chaque exemple, en expliquant qu'on cherche une suite strictement monotone variant vers le cas de base, car c'est facile de faire du récursif qui s'arrête pas. On peut parler un peu de preuve de correction s'il faut, mais encore plus avec les mains. Je pense qu'en rester aux "on voit bien que" et déplier des exemples suffira pour l'instant.

À propos des codes fournis : je pense qu'ils fonctionnent, mais je ne les ai pas testés, en fait...

Ce qui est important de faire :

- Les questions de base, avec récursivité simple. Il faut appliquer à la lettre la recette de cuisine du cours :
 - identifier sur quoi porte la récursion (ici, c'est tjs la longueur de la chaîne)
 - Identifier et résoudre les cas triviaux (ici, c'est souvent quand la chaîne est vide, plus de temps en temps quand le premier est ce qu'on cherche)

- Faire le cas général, ie faire le pb pour la chaîne complete en supposant que “quelqu’un” sait faire quand la chaîne est plus courte.
- Des questions où y’a une remontée récursive plus intelligente, ie, tous ceux qui font des *adj()* avec la récursion sur l’un de ses arguments.
- Des questions avec précondition. On se prend pas la tête, on l’écrit juste pour signifier “si quelqu’un est assez bête pour appeler la fonction dans un cas où c’est pas respecté, ça va mal se passer”. Pas besoin de vérifier explicitement la longueur de la chaîne, par exemple. On indique juste “Précondition : la chaîne est assez longue”.
- Des questions où y’a besoin d’un helper pour aller plus vite. Par exemple *nderniers* (dans sa 2ieme version) ou *retourne*. **C’est important.**
- Dans un monde parfait, il faut tout faire jusqu’à concat.

Ce qu’il est important de dire :

- Insister sur la terminaison (même si on le fait avec les mains). Ça termine car la longueur de la chaîne est strictement décroissante et que j’ai un cas terminal pour $lgr=0$. Il faut aussi dire que décroissante + cas terminal en 0 est pas assez : si on décroît de 2 en 2 en partant d’un impair, on “passe à travers”. Mais c’est pas le cas ici. Faire sentir tout ça, même si on démontre rien.
- Le coût algorithmique de chaque fonction. Souvent $\Theta(n)$, parfois $O(n)$, parfois autre.
- Insister sur l’intérêt des fonctions Helper, et comment on les construit : les arguments supplémentaires sont des accumulateurs dans lesquels le résultat se construit peu à peu (exemple de *retourne* ou *concat*). Ou alors dans lesquels une donnée précalculée est stockée (exemple de *Nderniers*).

Fin réponse

▷ **Question 1:** *longueur* : $\begin{cases} \text{List[Char]} \mapsto \text{Int} \\ \text{retourne le nombre de lettres composant la chaîne} \end{cases}$

Réponse

```

1 def longueur(l:List[Char]):Int = {
2   if (l == Nil)
3     return 0
4   else
5     return 1 + longueur(l.tail)
6 }

```

Idée pour trouver comment faire Imaginez que vous voulez savoir combien de camions sont devant vous sur cette petite route de montagne. Vous les voyez pas tous.

- Seule solution, vous en doublez 1, et vous savez qu’au total, il y en avait 1+ ce qui vous reste à doubler.
- Vous en doublez un autre, et au total, il y en avait 1+1+ce qui vous reste à doubler.
- Le jour où vous avez plus de camion devant vous, il vous en reste 0 à doubler, et au total, il y en avait 1+1+1+1+...+1+0.

Terminaison : La longueur est strictement décroissante et on s’arrête quand la chaîne est vide.

Correction : On se contente de déplier les appels au tableau aujourd’hui. Mais il faut le faire pour qu’ils comprennent, et il faut le faire à chaque fois, pas que celui là. **Pas cette année, j’ai pas présenté les preuves de prog encore. L’an prochain ça sera dans le bon ordre.**

Complexité : On regarde chaque lettre, on a donc $O(n)$ appels récursifs. A chaque appel, on n’appelle que des fonctions de base, pas chères. Donc une étape est en $O(1)$. Résultat : $O(n) \times O(1) = O(n)$

On peut l’écrire de façon plus jolie en Scala :

```

1 écriture fonctionnelle (sans return)
2 def longueur(l:List[Char]):Int = {
3   if (l == Nil)
4     0
5   else
6     1 + longueur(l.tail)
7 }

```

```

1 avec un zoli filtrage
2 def longueur(l:List[Char]):Int = {
3   l match {
4     case a::b => 1 + longueur(b)
5     case _    => 0
6   }
7 }

```

Fin réponse

▷ **Question 2:** $est_membre : \begin{cases} List[Char] \times Char \mapsto Boolean \\ \text{retourne true ssi le caractère fait partie de la chaîne} \end{cases}$

Réponse

<pre> 1 def isMember(l:List[Char], 2 value:Char):Boolean= { 3 if (l==Nil) false 4 else if (l.head == value) true 5 else isMember(l.tail, value) 6 } 7 </pre>	<pre> 1 def isMember(l:List[Char], v:Char):Boolean= { 2 1 match { 3 case a::_ if a==v => true 4 case a::b => isMember(b, v) 5 case _ => false 6 } 7 } </pre>
--	--

Est ce que cette fonction traite correctement le cas où le caractère n'est pas dedans.

Appliquez le meme algo à la recherche d'une peau rouge dans un oignon. Je regarde la peau extérieure, elle est pas rouge, je l'enlève et recommence. Je recommence sur toutes les peaux (toutes jaunes) jusqu'à la toute dernière. Je l'enlève aussi car elle est jaune. Je me retrouve avec l'oignon vide entre les mains, j'ai donc l'assurance qu'aucune peau n'était rouge dans mon oignon.

Terminaison et Complexité : comme avant. Simplement, chaîne vide n'est pas le seul cas terminal, mais ça ne gêne pas la terminaison. On pourrait chercher à faire une étude plus précise de la complexité avec meilleur des cas et pire des cas, mais ce n'est pas la peine. En moyenne, cet algo est linéaire. Faut juste leur faire sentir la complexité et laisser au module "Mat Num" le plaisir de faire des maths.

Complexité meilleur des cas c'est si la chaîne est vide ou qu'on cherche 't' dans 'toto'. $O(1)$

Complexité pire des cas Je cherche 'e' dans 'toto', je dois parcourir toute la chaîne. $O(n)$

Complexité cas moyen Ben on peut pas répondre avec si peu de données. Ceux qui répondent $\left[\frac{n}{2}\right]$ supposent l'équiprobabilité des lettres, c'est une hypothèse forte que l'on a pas. Imaginez chercher le β (on le z) dans la langue française, par rapport au 'e'.

Fin réponse

▷ **Question 3:** $occurrence : \begin{cases} List[Char] \times Char \mapsto Int \\ \text{retourne le nombre d'occurrences du caractère dans la chaîne} \end{cases}$

Réponse

occurrences(ch,c)	
<pre> 1 si ch = chvide alors 0 2 sinon si premier(ch) = c alors 1 + occurrence(reste(ch),c) 3 sinon occurrence(reste(ch),c) </pre>	<pre> 1 def occ(l:List[Char], value:Char):Int= { 2 if (l==Nil) 0 3 else if (l.head == value) 4 1+occ(l.tail,value) 5 else occ(l.tail, value) 6 } 7 </pre>
<pre> 1 def occ(l:List[Char], v:Char):Int= { 2 1 match { 3 case a::b if a==v => 1+occ(b, v) 4 case a::b => occ(b, v) 5 case _ => 0 6 } 7 } </pre>	

Terminaison et Complexité : comme avant : $O(n)$.

Fin réponse

▷ **Question 4:** $tous_différents : \begin{cases} List[Char] \mapsto Boolean \\ \text{retourne true ssi tous les membres de la chaîne sont différents} \end{cases}$

Réponse

tous_différents(ch)	
<pre> 1 si ch = chvide alors VRAI 2 sinon si est_membre(suite(ch), premier(ch)) alors FAUX 3 sinon tous_différents(suite(ch)) </pre>	
<pre> 1 def allDifferent(l:List[Char]):Boolean = { 2 if (l == Nil) true 3 else if (isMember(l.tail, l.head)) false 4 else allDifferent(l.tail) 5 } </pre>	

Terminaison : comme avant.

Complexité : On fait toujours $O(n)$ appels récursifs, mais ce coup ci, chacun fait appel à `est_membre`, qui est elle même en $O(n)$. Donc $C = O(n) \times O(n) = O(n^2)$

On peut se poser la question de l'optimalité. Est ce que c'est comme ca que vous vérifiez que toutes les cartes d'un paquet sont différentes ?? Non, bien sur. Le plus simple à la main, c'est de trier la pile dans un ordre donné, puis de faire un seul parcours en comparant chaque carte à la suivante (un peu comme la fonction `croissante`, donnée plus bas).

Étant donné qu'il existe des algos de tris en $O(n \times \log(n))$, on a

$$C = C_{\text{pretraitement}} + C_{\text{fonctionrecursive}} = O(n \times \log(n)) + O(n) = O(n \times \log(n))$$

. Ce qui est bien mieux que $O(n^2)$ quand n est grand.

Notons cependant que la complexité dans le meilleur des cas passe de $O(1)$ (quand la chaîne commence par 'aa', l'algo $O(n^2)$ répond immédiatement) à $O(n \times \log(n))$... sauf si on a fait son tri avec attention.

Fin réponse

▷ **Question 5:** `supprime` : $\begin{cases} \text{List}[\text{Char}] \times \text{Char} \mapsto \text{List}[\text{Char}] \\ \text{retourne la chaîne privée de la première occurrence du caractère.} \end{cases}$
Si le caractère ne fait pas partie de la chaîne, celle-ci est inchangée.

Réponse

```

1  si ch = chvide alors ch
2      sinon si premier(ch) = c alors reste(ch)
3      sinon adj(premier(ch), supprime(suite(ch),c))

1  def remove(l:List[Char], v:Char):List[Char]={
2      l match {
3          case a::b if a==v => remove(b, v)
4          case a::b         => a::remove(b,v)
5          case _             => Nil
6      }
7  }
```

Terminaison et Complexité : comme avant : $O(n)$.

Fin réponse

▷ **Question 6:** `deuxieme` : $\begin{cases} \text{List}[\text{Char}] \mapsto \text{Char} \\ \text{retourne le deuxième caractère de la chaîne} \end{cases}$

Réponse

```

1  // PRECONDITION: list != Nil et ch.tail != Nil
2  def second(list: List[Char]):Char = list.tail.head
```

Terminaison : C'est un appel direct, sans récursion. Mais c'est l'occasion de parler des préconditions.

On ne vérifie pas explicitement si la précondition est respectée dans le code, car on ne saurait pas trop comment réagir si elle est violée. On laisse donc Scala se débrouiller, il va lever une exception comme il se doit.

Oui, au passage, les accolades sont optionnelles en Scala quand on définit une fonction.

Complexité : $O(1)$

Fin réponse

▷ **Question 7:** `dernier` : $\begin{cases} \text{List}[\text{Char}] \mapsto \text{Char} \\ \text{retourne le dernier caractère de la chaîne} \end{cases}$

Réponse

```

1 // PRECONDITION: list != Nil
2 def last(list: List[Char]): Char = {
3   if (list.tail == Nil) list.head
4   else last(list.tail)
5 }

```

Terminaison et Complexité : comme avant : $O(n)$.

Fin réponse

▷ **Question 8:** *saufdernier* : $\begin{cases} \text{List[Char]} \mapsto \text{List[Char]} \\ \text{retourne la chaîne privée de son dernier caractère} \end{cases}$

Réponse

```

1 // PRECONDITION: list != Nil
2 def butLast(list: List[Char]): List[Char] = {
3   if (list.tail == Nil) Nil
4   else list.head :: butLast(list.tail)
5 }

```

Terminaison et Complexité : comme avant : $O(n)$.

Fin réponse

▷ **Question 9:** *nieme* : $\begin{cases} \text{List[Char]} \times \text{Int} \mapsto \text{Char} \\ \text{retourne le nieme caractère de la chaîne} \end{cases}$

Réponse

```

1 // PRECONDITION: list != Nil
2 def nth(list: List[Char], rank: Int): Char = {
3   if (rank==0) list.head
4   else nth(list.tail, rank-1)
5 }

```

Terminaison et Complexité : comme avant : $O(n)$.

Fin réponse

▷ **Question 10:** *npremiers* : $\begin{cases} \text{List[Char]} \times \text{Int} \mapsto \text{List[Char]} \\ \text{retourne les n premiers caractères de la chaîne} \end{cases}$

Réponse

```

1 // PRECONDITION: n>=list.size
2 def nFirsts(list: List[Char], amount: Int): List[Char] = {
3   if (amount == 0) Nil
4   else list.head :: nFirsts(list.tail, amount-1)
5 }

```

Terminaison et Complexité : comme avant : $O(n)$.

Correction : C'est un bon exemple pour faire une preuve de correction :

- Précondition à l'étape n entraîne (récursivement) la précondition pour les étapes suivantes avec des n plus petits
- Le traitement dans le cas terminal (pour $n = 0$) assure la post-condition
- Le traitement lors de la remontée assure la post-condition

Fin réponse

▷ **Question 11:** *nderniers* : $\begin{cases} \text{List[Char]} \times \text{Int} \mapsto \text{List[Char]} \\ \text{retourne les n derniers caractères de la chaîne} \end{cases}$

Réponse

Plusieurs variantes sont possibles :

```

1  def nLasts(list: List[Char], amount: Int): List[Char] = {
2    if (amount == list.size) list
3    else nLasts(list.tail, n-1)
4  }

```

Complexité : On a $O(n)$ appels récursifs, mais chacun fait un appel à longueur, qui est elle même en $O(n)$. Donc, $C = O(n) \times O(n) = O(n^2)$.

```

1  def nLasts(l: List[Char], n: Int): List[Char] = reverse(nFirst(reverse(list), n))

```

Complexité : On ajoute les complexités respectives de chaque appel. $C = O(n) + O(n) + O(n) = O(n)$, (car $C_{retourne} = O(n)$) ce qui est mieux. Mais `retourne` n'est pas encore défini, ce qui donne l'impression de tricher un peu. Alors on fait une troisième version, qui est surtout l'occasion d'introduire les fonctions helpers.

```

1  def nLasts(list: List[Char], n: Int): List[Char] = {
2    butNFirsts(list, list.size - n)
3
4    def butNFirst(list: List[Char], n: Int): List[Char] = {
5      if (n == 0 || list == Nil) list
6      else butNFirst(list.tail, n - 1)
7    }
8  }

```

L'idée est donc de calculer une bonne fois pour toute combien de caractères il faut retirer, puis de le faire ensuite sans réfléchir au lieu de (comme dans la première) regarder après chaque retrait si on en a enlevé assez. Ça permet de tomber la complexité en $O(n)$.

Fin réponse

▷ **Question 12:** `retourne` : $\begin{cases} \text{List[Char]} \mapsto \text{List[Char]} \\ \text{retourne la chaîne lue en sens inverse} \end{cases}$

Réponse

Cette fonction est très importante. Si vous manquez de temps, faites sauter d'autres fonction pour parvenir à faire celle là car on en a très besoin dans le TP2. Là encore, il y a plusieurs solutions.

```

1  def reverse(l: List[Char]): List[Char] = {
2    if (l == Nil) Nil
3    else last(l) :: reverse(butLast(l))
4  }

```

Complexité : $O(n)$ appels, et $O(n)$ chaque à cause de `saufdernier` et `dernier`. $O(n^2)$, donc.

Comment leur faire trouver mieux : Demandez leur de réfléchir à comment ils inversent l'ordre d'une pile de cartes, ou une pile d'assiettes : on prend une pile supplémentaire, on passe le premier de la pile de départ sur l'autre, et on recommence avec la deuxième de la pile de départ. Si ça aide pas, faut

détailler un exemple : A trier ABC

ABC	∅
BC	A
C	BA
∅	CBA

~~ résultat = CBA

```

1  def retourne(l: List[Char]): List[Char] = {
2
3    def helper(todo: List[Char], done: List[Char]): List[Char] = {
4      if (todo == Nil) done
5      else helper(todo.tail, todo.head :: done)
6    }
7
8    helper(l, Nil)
9  }

```

Comme souvent avec les fonctions helpers, on construit dans un argument supplémentaire le résultat final. Donc, on prend le travail qu'on aurait fait pendant la remontée, et on le fait dans la descente sur cet accumulateur. C'est important car ça change la fonction en récursive terminale (même s'ils n'ont pas encore vu ce que c'est à ce moment du cours).

Ce qui nous intéresse ici, c'est que la complexité passe en $O(n)$. Il est très important de déplier le retournement d'une chaîne d'exemple avec cette méthode.

Fin réponse

▷ **Question 13:** $concat : \begin{cases} List[Char] \times List[Char] \mapsto List[Char] \\ \text{retourne les deux chaînes concaténées} \end{cases}$

Réponse

version brutale: $O(n^2)$

```

1 def concat1(ch1:List[Char], ch2:List[Char]): List[Char] = {
2   if (ch1 == Nil) return ch2
3   return concat1(butLast(1), last(ch1)::ch2)
4 }
```

Pour aller plus vite, il faut mettre ch1 à l'envers une bonne fois pour toute au lieu d'aller piocher le dernier à tout bout de champ. On passe de $O(n^2)$ à $O(n)$, tout de même. Encore une fois, un exemple donné avant les aide à trouver tous seuls.

ABC	DEF	en donnée	
CBA	DEF	on inverse ch1 avant d'appeler helper	
BA	CDEF	réursion dans helper	↔ résultat = ABCDEF
A	BCDEF	réursion dans helper	
∅	ABCFED	Cas terminal de la récursion dans helper	

version avec helper: $O(n)$

```

1 def concat(ch1:List[Char], ch2:List[Char]):List[Char] = {
2   def helper(ch1:List[Char], ch2:List[Char]):List[Char] = {
3     if (ch1 == Nil) return ch2
4     else return helper(ch1.tail, ch1.head :: ch2)
5   }
6   helper(reverse(ch1), ch2)
7 }
```

On peut même se rendre compte que notre helper est exactement le même que celui de reverse. Du coup, on peut réécrire ainsi :

version avec helper: $O(n)$

```

1 def concat(ch1:List[Char], ch2:List[Char]):List[Char] = {
2   def helper(ch1:List[Char], ch2:List[Char]):List[Char] = {
3     if (ch1 == Nil) return ch2
4     else return helper(ch1.tail, ch1.head :: ch2)
5   }
6   helper(helper(ch1, Nil), ch2)
7 }
```

Fin réponse

▷ **Question 14:** $min_ch : \begin{cases} List[Char] \mapsto Char \\ \text{retourne le caractère le plus petit de la chaîne} \end{cases}$

On considère l'ordre lexicographique, et on suppose l'existence d'une fonction $min(a,b)$.

Réponse

$min_ch(ch)$

```

1 PRECONDITION: ch != chvide
2 def min(l:List[Char]): Char = {
3   def min2(l:List[Char], v:Char): Char = {
4     if (l==Nil) return v
5     if (l.head < v) return min2(l.tail, l.head)
6     return min2(l.tail, v)
7   }
8   return min2(l.tail, l.head)
9 }
```

Terminaison et Complexité : comme avant : $O(n)$.

Fin réponse

▷ **Question 15:** *croissante* : $\begin{cases} \text{List}[\text{Char}] \mapsto \text{booléen} \\ \text{retourne si la chaîne est croissante (dans l'ordre lexicographique)} \end{cases}$

Réponse

```

1 def increasing(l:List[Int]): Boolean = {
2   if (l == Nil || l.tail == Nil) return true
3   if (l.head > l.tail.head) return false
4   return increasing(l.tail)
5 }
```

Terminaison et Complexité : comme avant : $O(n)$.

Fin réponse

▷ **Question 16:** *nnaturels* : $\begin{cases} \text{Int} \mapsto \text{List}[\text{Char}] \\ \text{retourne une chaîne formée des } n \text{ premiers entiers naturels} \end{cases}$
 Dans un premier temps, on construira $\{n, n-1, n-2, \dots, 3, 2, 1\}$ avant de construire $\{1, 2, 3, \dots, n\}$.

Réponse

Version simple qui donne la liste à l'envers

```

1 nnaturels1(n):
2   si n = 0 alors chvide
3   sinon adj(n, nnaturels1(n-1))
```

Version trichée qui donne la chaîne à l'endroit:

```

1 nnaturels2(n):
2   retourne(nnaturels1(ch))
```

Pour faire la série dans l'ordre sans tricher, il faut une fonction d'aide. Pour le faire trouver, on peut écrire au tableau les différents arguments pris par cette fonction d'aide.

Version avec helper:

```

1 nnaturels3(n):
2   nnaturels3_helper(1,n-1)
3
4 nnaturels3_helper(n, todo):
5   si todo = 0 alors chvide
6   sinon adj(n, nnaturels3_helper(n+1,todo-1))
```

Fin réponse

▷ **Question 17:** *palindrome* : $\begin{cases} \text{List}[\text{Char}] \mapsto \text{booléen} \\ \text{retourne VRAI si la chaîne est un palindrome} \end{cases}$

Un palindrome se lit indifféremment de droite à gauche ou de gauche à droite. Exemple : « Esope reste et se repose ». On peut ignorer les espaces.

Réponse

palindrome(ch)

```

1 si longueur(ch) <= 1 alors
2   VRAI
3 sinon
4   si premier(ch) = dernier(ch) alors
5     palindrome(suite(saufdernier(ch)))
6   sinon
7     si premier(ch) = ' ' alors
8       palindrome(suite(ch))
9     sinon
10      si dernier(ch) = ' ' alors
11        palindrome(saufdernier(ch))
12      sinon
13        FAUX
14      finsi
15    finsi
16  finsi
17 finsi
```


La version simple est de ne pas ignorer les espaces, et de mettre un FAUX après le second «sinon» sans tester plus en avant.

Fin réponse

▷ **Question 18:** *anagramme* : $\left\{ \begin{array}{l} \text{List}[\text{Char}] \times \text{List}[\text{Char}] \mapsto \text{booléen} \\ \text{retourne VRAI si les chaînes sont des anagrammes l'une de l'autre} \end{array} \right.$

Une anagramme d'un mot est un autre mot obtenu en permutant les lettres. Exemples : «chien» et «niche»; «baignade» et «badinage»; «Sédution», «éconduits» et «on discute».

Réponse

```

1  si ch1=chvide et ch2=chvide alors
2      VRAI
3  sinon
4      si est_membre(premier(ch1), ch2) alors
5          anagramme(suivant(ch1), supprime(premier(ch1), ch2))
6      sinon
7          FAUX
8      finsi
9  finsi

```

Fin réponse

▷ **Question 19:** *union* : $\left\{ \begin{array}{l} \text{List}[\text{Char}] \times \text{List}[\text{Char}] \mapsto \text{List}[\text{Char}] \\ \text{retourne une chaîne formée de toutes les lettres de ch1 et ch2, sans doublons} \end{array} \right.$

On peut supposer dans un premier temps que ch1 et ch2 ne contiennent pas de doublons.

Réponse

```

1  si ch1=chvide alors
2      si ch2=chvide alors
3          chvide
4      sinon
5          union(ch2,chvide) -- Pour virer les doublons de ch2
6      finsi
7  sinon
8      si est_membre(premier(ch1), suite(ch1)) ou est_membre(premier(ch1), ch2) alors
9          union(suite(ch1),ch2)
10     sinon
11         adj(premier(ch1),union(suite(ch1),ch2))
12     finsi
13 finsi

```

Fin réponse

▷ **Question 20:** *difference* : $\left\{ \begin{array}{l} \text{List}[\text{Char}] \times \text{List}[\text{Char}] \mapsto \text{List}[\text{Char}] \\ \text{retourne toutes les lettres de ch1 ne faisant pas partie de ch2} \end{array} \right.$

Réponse

```

1  si ch2 = chvide alors
2      ch1
3  sinon
4      si ch1 = chvide alors
5          chvide
6      sinon
7          si est_membre(premier(ch1),ch2) alors
8              difference(suite(ch1),ch2)
9          sinon
10             adj(premier(ch1), difference(suite(ch1),ch2))
11         finsi
12     finsi
13 finsi

```

Fin réponse