CATELLA Solène
LE BAHER Hugo

21/12/2018

Langages de programmation de haut-niveau
Préfixes - Julia

Ce projet, réalisé à partir du langage de programmation *Julia*, visait à dégager les principales informations statistiques liées à un texte, au niveau global (nombre de mots totaux et distincts le composant) comme local (longueur moyenne d'un mot).

L'intérêt de l'exercice ici proposé portait avant tout sur l'optimisation des structures de données employées pour représenter un texte en machine. Deux approches ont ici été explorées (I et II). Les comparaisons en termes d'empreinte mémoire et de temps d'exécution, présentées à la fin de ce rapport (III), soulignent toute l'importance du choix d'une structure de données bien adaptée.

I. Structures de données et fonctions implémentées

Comme évoqué plus haut, la représentation d'un texte en machine s'est faite via deux structures de données :

1. un tableau de *String*, sous forme de *struct*, comportant l'ensemble des mots distincts d'un texte, avec pour attributs :
    - *nbMots*, le nombre total de mots du texte
    - *nbMotsDistincts*, le nombre de mots distincts du texte
    - *mots*, tableau des mots distincts du texte
    - *decompte*, tableau des occurrences des mots du texte

2. un arbre des préfixes, étiqueté au niveau de ses arêtes par une suite de caractères (*Char*), qui, lorsque concaténées, restituent l'ensemble des différents mots d'un texte. Ses attributs sont :
    - *terminal*, indiquant si les caractères de l'arête parcourus jusqu'ici représentent un mot éligible
    - *nb*, le nombre de mots totaux qui sont définis sur l'arbre ou dans ses descendants

- *suite*, un tableau qui associe un caractère c à l'arbre des préfixes des mots qui commencent par c

À chacune de ces représentations ont été associées un ensemble respectif de fonctions, parmi lesquelles :

1. une fonction segmentant un texte donné en mots/caractères : *segmenterTexteTableau* et *segmenterTexteArbre ;*
2. une fonction comptabilisant le nombre de mots distincts d'un texte : *calculerNbMotsDistinctsArbre* (côté tableau, cette opération est incluse dans la fonction *segmenterTexteTableau*) ;
3. une fonction déterminant si un mot donné est présent ou non dans un texte : *verifierMotTableau* et *verifierMotArbre* ;
4. une fonction calculant la longueur moyenne des mots d'un texte : *calculerLongMoyMotTab* et *calculerLongMoyMotArbre* ;
5. une fonction retournant la liste des mots de préfixe donné : *chercherMotsPrefixeTab* et *chercherMotsPrefixeArbre* ;
6. une fonction retournant la liste des mots de suffixe donné : *chercherMotsSuffixeTab* et *chercherMotsSuffixeArbre*.

Deux fonctions supplémentaires ont été implémentées :

- une fonction retournant la liste des mots d'un texte après suppression d'un caractère donné : *convertirMotTab et motsSansLettre* ;
- une fonction retournant le mot d'un texte de score maximal au Scrabble : *scoreMaxMotTab* et *scoreMaxMotArbre*.

Les fonctions associées à la structure de données *Tableau* sont fournies dans le fichier *Tableau.jl* ; celles liées à la structure de données *Arbre* le sont dans le fichier *Arbre.jl*.
Ces différentes fonctions ont été testées dans le fichier *Test.jl*, disponible en Annexe de ce rapport. Les résultats retournés par ces fonctions ont été soigneusement vérifiés d'une structure de données à l'autre. Quelques exemples sont ici présentés :

Recherche du mot de score maximum au Scrabble dans le texte de Cyrano de Bergerac :

```
sol@sol-VirtualBox:~/Documents/Julia$ julia Test.jl
("HIPPOCAMPELEPHANTOCAMÉLOS", 47)
```

mot : "Hippocampelephantocamélos", score : 42

Recherche des mots ayant pour préfixe "ros" dans le texte du Petit Prince :

```
sol@sol-VirtualBox:~/Documents/Julia$ julia Test.jl
["ROSIER", "ROSIERS", "ROSE", "ROSES"]
```

Mots débutant par "ros" : rosier, rosiers, rose, roses

II. Problèmes soulevés et résolus

Un premier problème, lié au découpage des mots du texte, a été rencontré. Pour pouvoir gérer au mieux la division du texte en chaîne de caractères valide, une liste de 23 séparateurs de mots possibles a été mise au point de façon itérative. Dans la mesure où cette liste a été établie à partir des textes proposés, on peut imaginer qu'elle ne soit pas exhaustive, c'est-à-dire que des problèmes de découpage puissent émerger si elle était appliquée sur d'autres textes.

L'implémentation des fonctions exigées n'a soulevé aucun problème particulier. Seule la fonction retournant le mot d'un texte de score maximal au Scrabble a nécessité une opération de normalisation, de façon à comptabiliser/scorer les caractères accentués au même titre que les caractères non accentués. La fonction *Unicode.normalize(mot, stripmark=true)* a donc été ici utilisée.

L'évaluation du temps d'exécution avec "@time" nous renvoyait des temps incompréhensibles au vue de l'estimation de la complexité de nos fonctions. Par exemple, des fonctions récursives qui ne nécessitent qu'une dizaine d'appels en tout, étaient indiquées comme anormalement lentes. De plus, la succession de plusieurs lignes avec des mesures semble fausser les résultats. Nous avons donc utilisé le framework "BenchmarkTools" et la macro "@btime" qui remplit la même fonction, mais plus rigoureusement.

III. Comparaison des coûts en temps et en espace

La comparaison des fonctions implémentées en termes de coûts en temps et espace a été effectuée dans le fichier *Main.jl*. Ces coûts sont ici présentés pour chacune des fonctions implémentées au travers des deux structures de données. La première ligne correspond aux mesures effectuées sur le texte Cyrano de Bergerac ; la seconde sur le texte du Petit Prince. À titre de rappel, le texte Cyrano de Bergerac contient un total de 36 280 mots, dont 5 482 différents; celui du Petit Prince un total de 15 424 mots, dont 2 401 différents.

NB : les complexités théoriques ici données sont en O (complexités au pire).

1. Fonction de segmentation d'un texte

Tableau :
2.759 s (103431172 allocations: 4.62 GiB)
612.691 ms (22382777 allocations: 1022.81 MiB)

Arbre :
58.336 ms (834844 allocations: 38.83 MiB)
18.844 ms (332111 allocations: 15.79 MiB)

2. Fonction vérifiant si un mot donné est présent dans un texte

Tableau :
9.321 µs (5 allocations: 256 bytes)
4.021 µs (5 allocations: 256 bytes)

Arbre :
363.125 ns (5 allocations: 160 bytes)
239.334 ns (3 allocations: 96 bytes)

3. Fonction calculant la longueur moyenne d'un mot d'un texte

Tableau :
329.138 µs (10472 allocations: 292.22 KiB)

Arbre :
3.793 ms (1 allocation: 16 bytes)

4. Fonction retournant la liste des mots de préfixe donné

Tableau :
686.634 µs (27419 allocations: 1.34 MiB)
330.149 µs (12021 allocations: 601.02 KiB)

Arbre :
48.880 µs (958 allocations: 36.11 KiB)
5.733 µs (110 allocations: 3.98 KiB)

5. Fonction retournant la liste des mots de suffixe donné

Tableau :
941.743 µs (32898 allocations: 1.51 MiB)
348.071 µs (14427 allocations: 676.66 KiB)

Arbre :
6.782 ms (53536 allocations: 3.19 MiB)
2.783 ms (28647 allocations: 1.64 MiB)

6. Fonction retournant la liste des mots après suppression d'un caractère donné

Tableau :
2.274 ms (26849 allocations: 1.21 MiB)
953.809 µs (11099 allocations: 514.53 KiB)

Arbre :
13.475 ms (160334 allocations: 6.92 MiB)
5.864 ms (70944 allocations: 3.05 MiB)

7. Fonction retournant le mot d'un texte de score maximal au Scrabble

Tableau :
14.334 ms (71273 allocations: 3.08 MiB)
6.244 ms (31246 allocations: 1.35 MiB)

Arbre :
7.436 ms (106289 allocations: 3.73 MiB)
3.001 ms (47473 allocations: 1.67 MiB)

On observe que les résultats sont assez cohérents avec ce que l'on espérait. En effet, par exemple, pour tout parcours complet dans la structure de données, nous avons :

- une complexité de $O(n)$ pour le tableau dans tous les cas, où n est le nombre de mots distincts ;
- une complexité de $O(n)$ pour l'arbre seulement dans le pire des cas, où n est le nombre de mots distincts. Le pire des cas étant un arbre stockant des mots totalement différents sans préfixes en commun, cette structure s'apparenterait donc à une liste de listes chaînées des caractères des mots. Chaque préfixe en commun entre deux mots augmente la performance de la structure.

Il est donc normal d'avoir une performance meilleure pour les arbres dans les fonctions **1, 4 et 7** qui consistent à parcourir tous les mots. Cependant, nous devrions aussi avoir de meilleures performances dans les fonctions **3 et 6**.

On voit que la recherche d'un élément (**2**) est aussi meilleure pour l'arbre. Un tel algorithme consiste juste à suivre chaque caractère, de noeud en noeud. La complexité est donc en $O(m)$, soit m la longueur du mot recherché. Pour le tableau non trié, il faut normalement parcourir chaque élément et le comparer au mot recherché, une complexité en $O(n * m)$. Si l'arbre obtient de meilleurs résultats, on observe quand même que la différence n'est pas aussi importante qu'elle devrait l'être.

Enfin, la recherche d'un suffixe (**5**) ne bénéficie pas du tout de l'implémentation en arbre des préfixes. Naturellement, pour le tableau, l'algorithme est le même que pour la recherche d'un préfixe. Cependant pour l'arbre des préfixes, notre algorithme doit parcourir tous ses mots,

parfois plusieurs fois pour comparer. Même si notre algorithme n'est peut être pas le meilleur, il nous semble impossible de trouver une implémentation meilleure que celle des tableaux pour ce cas-ci.

Pour expliquer les différences obtenues entre nos études théoriques et le benchmark, nous avons deux pistes de réflexion :

- D'une part, les fonctions du tableau étant implémentées au maximum grâce aux fonctions et aux syntaxes destinées à la manipulation de tableaux, nous soupçonnons le compilateur d'optimiser grandement ces fonctions, ce qui expliquerait des résultats si rapides.
- D'autre part, dans la mesure où les implémentations des fonctions sur les arbres utilisent toutes des formes récursives, il semblerait que cela ait un impact négatif sur les performances.

Pour vérifier la première hypothèse, nous avons implémenté une fonction sans utiliser les raccourcis de manipulation de tableaux :

```
function chercherMotsPrefixeTabLENT(tableau, prefixe)
    p =  Unicode.normalize(uppercase(prefixe), stripmark=true)
    mots = []
    for i = 1:length(tableau.mots)
        mot = Unicode.normalize(uppercase(tableau.mots[i]),
stripmark=true)
        if(length(mot)>=length(p))
            j=1
            found = true
            while (j<=length(p) && found)
                if(mot[j] != p[j])
                    found = false
                end
                j += 1
            end
            if(found)
                push!(mots, mot)
            end
        end

    end
    return mots
end
```

Cette implémentation révèle la lenteur d'un tel parcours sans les optimisations du compilateur :

Version tableau : 1.517 ms (18706 allocations: 903.36 KiB)
Version tableau optimisée : 331.555 µs (12011 allocations: 600.52 KiB)
Version arbre : 5.143 µs (110 allocations: 3.98 KiB)

IV. Annexes

Fichier Tableau.jl

```
import Unicode
mutable struct TableauMots
	nbMots::Int64
	nbMotsDistincts::Int64
	mots::Array{String}
	decompte::Array{Int64}
	function TableauMots()
	return new(0,0,Array{String,1}(),Array{Int64,1}())
	end
end

#Base.show(io::IO, tab::TableauMots) = print(io, " $(tab.nbMots) words,
$(tab.nbMotsDistincts) distincts words")

function segmenterTexteTableau(texte)
	tabMots = TableauMots()
	sep = [' ',',',';','.','-
','_','\'','`','\"','/',')','(','{','}','[',']','=','+','@','!','?','%','
\t']
	flux = open(texte,"r")
	while ! eof(flux)
	ligne = readline(flux)
	spl = split(uppercase(ligne), sep)
	ajouterMotsTableau(tabMots, spl)
	end
	close(flux)
	return tabMots
end

function ajouterMotsTableau(tableau, ligne)
	for mot in ligne
	if length(mot) > 0
		if !in(mot, tableau.mots)
			push!(tableau.mots, mot)
			tableau.nbMotsDistincts += 1
	 push!(tableau.decompte, 1)
	else
	 index = findfirst(x -> x == mot, tableau.mots)
	 tableau.decompte[index] += 1
		end
		tableau.nbMots += 1
	end
```

```
      end
end

function verifierMotTableau(mot, tableau)
    return in(uppercase(mot), tableau.mots)
end

function calculerLongMoyMotTab(tableau)
    return round(100*(sum([length(mot)*tableau.decompte[i] for (i, mot)
in enumerate(tableau.mots)]))/tableau.nbMots)/100
end

function chercherMotsPrefixeTab(tableau, prefixe)
    return([mot for mot in tableau.mots if startswith(mot,
uppercase(prefixe))])
end

function chercherMotsSuffixeTab(tableau, suffixe)
    return([mot for mot in tableau.mots if endswith(mot,
uppercase(suffixe))])
end

### Fonctions supplémentaires ###

function convertirMotTab(tableau, caractere1)
    return [if occursin(uppercase(caractere1), mot) replace(mot,
uppercase(caractere1) => "") else mot end for mot in tableau.mots]
end

function scoreMaxMotTab(tableau, scrabbleDico)
    scoreMot = findmax([(sum(get(scrabbleDico, c, 0) for c in
uppercase(Unicode.normalize(mot, stripmark=true)))) for mot in
tableau.mots])
    return (tableau.mots[scoreMot[2]], scoreMot[1])
end
```

Fichier Arbre.jl

```
import Unicode

mutable struct ArbreMots
      terminal::Bool
      nb::Int64
      suite::Dict{Char,ArbreMots}
      function ArbreMots()
          return new(false,0,Dict{Char,ArbreMots}())
    end
end

Base.show(io::IO, a::ArbreMots) = print(io, " $(a.nb) words,
$(calculerNbMotsDistinctsArbre(a)) distincts words")

function segmenterTexteArbre(texte)
      arbreMots = ArbreMots()
    sep = [' ',',',';','.','-
','_','\'','`','\"','/',')','(','{','}','[',']','=','+','@','!','?','%',
'\t']
    flux = open(texte,"r")
    while ! eof(flux)
        ligne = readline(flux)
        spl = split(uppercase(ligne), sep)
            for mot in spl
                  if length(mot) > 0
                  ajouterMotArbre(arbreMots, mot)
                  end
            end
    end
    close(flux)
    return arbreMots
end

function ajouterMotArbre(arbre, mot)
      if mot == ""
            arbre.terminal = true
      else
            first = mot[1]
            if(!(first in keys(arbre.suite)))
                  arbre.suite[first] = ArbreMots()
            end
            ajouterMotArbre(arbre.suite[first], mot[nextind(mot, 1):end])
      end
      arbre.nb += 1
```

```
end

function verifierMotArbre(mot, arbre)
	if mot == ""
		return arbre.terminal
	else
		first = uppercase(mot[1])
		if(!(first in keys(arbre.suite)))
			return false
		end
		return verifierMotArbre(mot[nextind(mot, 1):end],
arbre.suite[first])
	end
end

function calculerLongMoyMotDistinctsArbre(arbre)
	nb, tailletotale = longueurTotaleMotsDistincts(arbre, 0)
	return round(100*(tailletotale/nb))/100
end

function longueurTotaleMotsDistincts(arbre, profondeur) # =>
nbmotsdistincts, taille totale
	tailletotale = arbre.terminal * profondeur
	nb = arbre.terminal
	for (k,fils) in arbre.suite
		n, t = longueurTotaleMotsDistincts(fils, profondeur+1)
		nb += n
		tailletotale += t
	end
	return (nb, tailletotale)
end

function calculerLongMoyMotArbre(arbre)
	nb, tailletotale = longueurTotaleMots(arbre, 0)
	return round(100*tailletotale/nb)/100
end

function longueurTotaleMots(arbre, profondeur) # => nbmots, taille
totale
	nbfils, tailletotale = 0, 0
	for (k,fils) in arbre.suite
		n, t = longueurTotaleMots(fils, profondeur+1)
		nbfils += n
		tailletotale += t
	end
	tailletotale += arbre.terminal * profondeur * ( arbre.nb - nbfils )
```

```
	return (arbre.nb, tailletotale)
end

function chercherMotsPrefixeArbre(arbre, prefixe)
	if(prefixe=="")
		mots = []
		if(arbre.terminal)
			push!(mots, "")
		end
		for (k,fils) in arbre.suite
			append!(mots, [string(k)*mot for mot in
chercherMotsPrefixeArbre(fils, "")])
		end
		return mots
	else
		first = uppercase(prefixe[1])
		if(!(first in keys(arbre.suite)))
			return []
		end
		return [string(first)*mot for mot in
chercherMotsPrefixeArbre(arbre.suite[first], prefixe[nextind(prefixe,
1):end])]
	end
end

function chercherMotsSuffixeArbre(arbre, suffixe)
	return cmsar(arbre, uppercase(suffixe), uppercase(suffixe))
end

function cmsar(arbre, cursuffixe, suffixe)
	mots = []
	if(cursuffixe=="")
		if(arbre.terminal)
			push!(mots, "")
		end
	else
		if(cursuffixe[1] in keys(arbre.suite))
			append!(mots, [cursuffixe[1:1]*mot for mot in
cmsar(arbre.suite[cursuffixe[1]], cursuffixe[nextind(cursuffixe,
1):end], suffixe)])
		end
	end

	if !(length(cursuffixe)>0 && length(cursuffixe)<length(suffixe))
		for (k,fils) in arbre.suite
			append!(mots, [string(k)*mot for mot in cmsar(fils,
```

```
suffixe, suffixe)])
            end
      end
      return mots
end

### Fonctions supplémentaires ###

function scoreMaxMotArbre(arbre, dico)
      (scoremax, motmax) = (0, "")
      if(length(arbre.suite)==0)
            return (scoremax, motmax)
      end
      for (k,fils) in arbre.suite
            if(all(isletter, string(k)))
                  score, mot = scoreMaxMotArbre(fils, dico)
                  kscore = dico[Unicode.normalize(string(k),
stripmark=true)[1]]

                  if((score+kscore)>scoremax)
                        (scoremax, motmax) = (score+kscore, k*mot)
                  end
            end
      end
      return (scoremax, motmax)
end

function calculerNbMotsDistinctsArbre(arbre)
      count = arbre.terminal
      for (k,fils) in arbre.suite
            count += calculerNbMotsDistinctsArbre(fils)
      end
      return count
end

function chercherMots(arbre)
      mots = []
      if(arbre.terminal)
            push!(mots, "")
      end
      for (k,fils) in arbre.suite
            append!(mots, [string(k)*mot for mot in chercherMots(fils)])
      end
      return mots
end
```

```
function motsSansLettre(arbre, c)
	mots = []
	if(arbre.terminal)
		push!(mots, "")
	end
	for (k,fils) in arbre.suite
		if(k==uppercase(c))
			append!(mots, [mot for mot in motsSansLettre(fils, c)])
		else
			append!(mots, [string(k)*mot for mot in
motsSansLettre(fils, c)])
		end
	end
	return mots
end

function frequenceMoyenneArbre(arbre)
	return
round(100*(arbre.nb/calculerNbMotsDistinctsArbre(arbre)))/100
end
```

Fichier Main.jl

```
using BenchmarkTools

include("./Arbre.jl")
include("./Tableau.jl")

texteCyrano = "./cyrano.txt"
textePetitPrince = "./le_petit_prince.txt"

# Segmentation du texte #
cyranotab = segmenterTexteTableau(texteCyrano)
princetab = segmenterTexteTableau(textePetitPrince)
cyranoarb = segmenterTexteArbre(texteCyrano)
princearb = segmenterTexteArbre(textePetitPrince)

scrabbleDico = Dict('A' => 1, 'E' => 1, 'I' => 1, 'O' => 1, 'U' => 1, 'L'
=> 1, 'N' => 1, 'R' => 1, 'S' => 1, 'T' => 1, 'D' => 2, 'G' => 2, 'B' =>
3, 'C' => 3, 'M' => 3, 'P' => 3, 'F' => 4, 'H' => 4, 'V' => 4, 'W' => 4,
'Y' => 4, 'K' => 5, 'J' => 8, 'X' => 8, 'Q' => 10, 'Z' => 10)

println("Chargement et segmentation des fichiers")
println("   Tableau :")
@btime cyranotab = segmenterTexteTableau(texteCyrano)
@btime princetab = segmenterTexteTableau(textePetitPrince)

println("   Arbre :")
@btime cyranoarb = segmenterTexteArbre(texteCyrano)
@btime princearb = segmenterTexteArbre(textePetitPrince)

# Détection de mot dans un texte #
println("Detection existence")
println("   Tableau :")
@btime verifierMotTableau("Jaloux", cyranotab)
@btime verifierMotTableau("rose", princetab)

println("   Arbre :")
@btime verifierMotArbre("Jaloux", cyranoarb)
@btime verifierMotArbre("rose", princearb)

# Calcul de la longueur moyenne d'un mot dans un texte
println("Calcul longueur moyenne des mots")
println("   Tableau :")
@btime calculerLongMoyMotTab(cyranotab)
@btime calculerLongMoyMotTab(princetab)
```

```
println("   Arbre :")
@btime calculerLongMoyMotArbre(cyranoarb)
@btime calculerLongMoyMotArbre(princearb)

# Liste des mots commençant par une chaine de caractères donnée
println("Recherche préfixes")
println("   Tableau :")
@btime chercherMotsPrefixeTab(cyranotab, "am")
@btime chercherMotsPrefixeTab(princetab, "ros")

println("   Arbre :")
@btime chercherMotsPrefixeArbre(cyranoarb, "am")
@btime chercherMotsPrefixeArbre(princearb, "ros")

# Liste des mots terminant par une chaine de caractères donnée
println("Recherche suffixes")
println("   Tableau :")
@btime chercherMotsSuffixeTab(cyranotab, "acher")
@btime chercherMotsSuffixeTab(princetab, "eur")

println("   Arbre :")
@btime chercherMotsSuffixeArbre(cyranoarb, "acher")
@btime chercherMotsSuffixeArbre(princearb, "eur")


println("Recherche du mot de score maximal au Scrabble")
println("   Tableau :")
@btime scoreMaxMotTab(cyranotab, scrabbleDico)
@btime scoreMaxMotTab(princetab, scrabbleDico)

println("   Arbre :")
@btime scoreMaxMotArbre(cyranoarb, scrabbleDico)
@btime scoreMaxMotArbre(princearb, scrabbleDico)

println("Mots du tableau avec suppression d'un caractère")
println("   Tableau :")
@btime convertirMotTab(cyranotab, 'e')
@btime convertirMotTab(princetab, 'e')

println("   Arbre :")
@btime motsSansLettre(cyranoarb, 'e')
@btime motsSansLettre(princearb, 'e')
```

Fichier Test.jl

```
include("./Arbre.jl")
include("./Tableau.jl")

texteCyrano = "./cyrano.txt" # 36 280 mots dont 5 482 différents
textePetitPrince = "./le_petit_prince.txt" # 15 424 mots dont 2 401
différents

scrabbleDico = Dict('A' => 1, 'E' => 1, 'I' => 1, 'O' => 1, 'U' => 1, 'L'
=> 1, 'N' => 1, 'R' => 1, 'S' => 1, 'T' => 1, 'D' => 2, 'G' => 2, 'B' =>
3, 'C' => 3, 'M' => 3, 'P' => 3, 'F' => 4, 'H' => 4, 'V' => 4, 'W' => 4,
'Y' => 4, 'K' => 5, 'J' => 8, 'X' => 8, 'Q' => 10, 'Z' => 10)

## TESTS ##

## Segmentation du texte
cyranotab = segmenterTexteTableau(texteCyrano) # Version Tableau
princetab = segmenterTexteTableau(textePetitPrince)

cyranoarb = segmenterTexteArbre(texteCyrano) # Version Arbre
princearb = segmenterTexteArbre(textePetitPrince)

println(cyranotab.nbMots)
println(cyranotab.nbMotsDistincts)

println(princetab.nbMots)
println(princetab.nbMotsDistincts)

## Détection d'un mot dans un texte
println(verifierMotTableau("jaloux", cyranotab)) # Version Tableau
println(verifierMotTableau("soleil", princetab))

println(verifierMotArbre("jaloux", cyranoarb)) # Version Arbre
println(verifierMotArbre("soleil", princearb))

## Calcul de la longueur moyenne des mots d'un texte
println(calculerLongMoyMotTab(cyranotab)) # Version Tableau
println(calculerLongMoyMotTab(princetab))

println(calculerLongMoyMotArbre(cyranoarb)) # Version Arbre
println(calculerLongMoyMotArbre(princearb))

## Mots débutant par une chaîne de caractères donnée (préfixe)
println(chercherMotsPrefixeTab(cyranotab, "am")) # Version Tableau
println(chercherMotsPrefixeTab(princetab, "ros"))
```

```
println(chercherMotsPrefixeArbre(cyranoarb, "am")) # Version Arbre
println(chercherMotsPrefixeArbre(princearb, "ros"))

## Mots terminant par une chaîne de caractères donnée (suffixe)
println(chercherMotsSuffixeTab(cyranotab, "acher")) # Version Tableau
println(chercherMotsSuffixeTab(princetab, "eur"))

println(chercherMotsSuffixeArbre(cyranoarb, "acher")) # Version Arbre
println(chercherMotsSuffixeArbre(princearb, "eur"))

## Liste des nouveaux mots après suppression d'un caractère
println(convertirMotTab(cyranotab, "p")) # Version Tableau
println(convertirMotTab(princetab, "p"))

println(motsSansLettre(cyranoarb, "p")) # Version Arbre
println(motsSansLettre(princearb, "p"))

## Calcul du mot de score maximal au Scrabble d'un texte
println(scoreMaxMotTab(cyranotab, scrabbleDico)) # Version Tableau
println(scoreMaxMotTab(princetab, scrabbleDico))

println(scoreMaxMotArbre(cyranoarb, scrabbleDico)) # Version Arbre
println(scoreMaxMotArbre(princearb, scrabbleDico))
```