TD2 : Récursivité et chaînes récursives

TOP : Techniques and tOols for Programming – 1A

★ **Exercice 1:** Code mystère.

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

▷ **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 ?

Notez que tout le travail est fait par la fonction g, 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 fibonnacci, 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 plutot 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” (meme 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).

Fin réponse

★ **Exercice 2:** Soit le type `Chaine` muni des opérations suivantes :

$$\begin{cases} chvide : \emptyset \mapsto Chaine & \\ premier : Chaine \mapsto Caractère & \text{(défini ssi la chaîne n'est pas vide)} \\ reste : Chaine \mapsto Chaine & \text{(défini ssi la chaîne n'est pas vide)} \\ adj : Chaine \times Caractère \mapsto Chaine & \end{cases}$$

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”. Et ensuite, on prend les vrais débutants par la main.

A propos des preuves Je suis désolé, vu que j'ai 2 séances cette semaine et pis plus rien pendant un grand moment, et pleins de TD/TP entre temps, j'ai pas eu le temps de parler de preuve de programme, encore. Du coup, il faut leur expliquer avec les mains ce qu'on veut faire à propos de la terminaison. Pourquoi c'est important et comment on le montre... Laissez tomber la correction, sauf pour dire “on voit bien” et déplier des exemples quand le premier argument suffit pas.

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 chaine)
  - Identifier et résoudre les cas triviaux (ici, c'est souvent quand la chaine 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 chaine complete en supposant que “quelqu'un” sait faire quand la chaine 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 tete, on l'écrit juste pour signifier “si quelqu'un est assez bête pour appeller la fonction dans un cas où c'est pas respecté, ca va mal se passer”. Pas besoin de vérifier explicitement la longueur de la chaine, par exemple. On indique juste “Précondition : la chaine 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 (meme si on le fait avec les mains). Ca termine car la longueur de la chaine 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 décroit de 2 en 2 en partant d'un impaire, on "passe à travers". Mais c'est pas le cas ici.
  Faire sentir tout ca, meme 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* : $\begin{cases} Chaine \mapsto \mathbb{N} \\ \text{retourne le nombre de lettres composant la chaîne} \end{cases}$

Réponse

longueur(ch)

```
si ch = chvide alors 0
               sinon 1 + longueur(reste(ch))
```

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 doubler un autre, et au total, il y en avait 1+1+ce qui vous reste à doubler.
- Le jour ou 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 ca 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 : $O(n) \times O(1) = O(n)$

Fin réponse

▷ **Question 2:** *est_membre* : $\begin{cases} Chaine \times caract\grave{e}re \mapsto bool\acute{e}en \\ \text{retourne VRAI ssi le caractère fait partie de la chaîne} \end{cases}$

Réponse

est_membre(ch,c)

```
si ch = chvide alors FAUX
               sinon si premier(ch) = c alors VRAI
                                        sinon est_membre(reste(ch),c)
```

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, chaine vide n'est pas le seul cas terminal, mais ca ne gène 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 chaine 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 chaine. $O(n)$

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

Fin réponse

▷ **Question 3:** *occurence* : $\left\{ \begin{array}{l} Chaine \times caractère \mapsto \mathbb{N} \\ \text{retourne le nombre d'occurences du caractère dans la chaine} \end{array} \right.$

Réponse

occurences(ch,c)

```
si ch = chvide alors 0
            sinon si premier(ch) = c alors 1 + occurence(reste(ch),c)
                                     sinon     occurence(reste(ch),c)
```

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

Fin réponse

▷ **Question 4:** *tous_differents* : $\left\{ \begin{array}{l} Chaine \mapsto booléen \\ \text{retourne VRAI ssi tous les membres de la chaine sont différents} \end{array} \right.$

Réponse

tous_differents(ch)

```
si ch = chvide alors VRAI
            sinon si est_membre(suite(ch), premier(ch)) alors FAUX
                                              sinon tous_differents(suite(ch))
```

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 différente ? ? 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

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

. 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* : $\left\{ \begin{array}{l} Chaine \times caractère \mapsto Chaine \\ \text{retourne la chaine privée de la première occurence du caractère.} \end{array} \right.$

Si le caractère ne fait pas partie de la chaîne, celle-ci est inchangée.

Réponse

supprime(ch,c)

```
si ch = chvide alors ch
            sinon si premier(ch) = c alors reste(ch)
                                     sinon adj(premier(ch), supprime(suite(ch),c))
```

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

Fin réponse

▷ **Question 6:** $deuxieme : \begin{cases} Chaine \mapsto caractère \\ \text{retourne le deuxième caractère de la chaîne} \end{cases}$

Réponse

deuxieme(ch)
```
PRECONDITION: ch != vide et suite(ch) != vide
premier(suite(ch))
```

Terminaison : C'est un appel direct, sans récursion. Mais c'est l'occasion de réintroduire les préconditions.
Complexité : O(1)

Fin réponse

▷ **Question 7:** $dernier : \begin{cases} Chaine \mapsto caractère \\ \text{retourne le dernier caractère de la chaîne} \end{cases}$

Réponse

dernier(ch)
```
PRECONDITION: ch != vide
si suite(ch) = vide alors premier(ch)
                    sinon dernier(suite(ch))
```

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

Fin réponse

▷ **Question 8:** $saufdernier : \begin{cases} Chaine \mapsto Chaine \\ \text{retourne la chaine privée de son dernier caractère} \end{cases}$

Réponse

saufdernier(ch)
```
PRECONDITION: ch != chvide
si suite(ch) = chvide alors chvide
                      sinon adj(premier(ch), saufdernier(suite(ch)))
```

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

Fin réponse

▷ **Question 9:** $nieme : \begin{cases} Chaine \times \mathbb{N} \mapsto caractère \\ \text{retourne le nieme caractère de la chaîne} \end{cases}$

Réponse

nieme(ch,n)
```
PRECONDITION: ch != vide
si n = 0 alors alors premier(ch)
          sinon nieme(suite(ch), n-1)
```

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

Fin réponse

▷ **Question 10:** $npremiers : \begin{cases} Chaine \times \mathbb{N} \mapsto Chaine \\ \text{retourne les n premiers caractères de la chaîne} \end{cases}$

Réponse

npremiers(ch,n)
```
PRECONDITION: n>=longueur(ch)
si n = 0 alors chvide
         sinon adj(premier(ch), npremiers(n-1,suite(ch)))
```

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

CORRECTION

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

- Précondition à l'étape n entraine (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* : $\left\{ \begin{array}{l} Chaine \times \mathbb{N} \mapsto Chaine \\ \text{retourne les n derniers caractères de la chaîne} \end{array} \right.$

Réponse

Plusieurs variantes sont possibles :

nderniers (ch,n) -- version naive
```
ndernier(ch,n) = si lgr(ch)=n alors ch
                 sinon ndernier(reste(ch), n)
```

Complexité : On a $O(n)$ appels récurssifs, mais chacun fait un appel à longueur, qui est elle même en $O(n)$. Donc, $C = 0(n) \times O(n) = O(n^2)$.

nderniers (ch,n) -- avec retourne
```
ndernier(ch,n) = retourne(npremiers(retourne(ch), 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. Alors on en fait une troisième qui est l'occasion d'introduire les fonctions helpers.

nderniers (ch,n) -- avec fonction d'aide
```
ndernier(ch,n) = supp_n_premiers(ch,lgr(ch)-n)

supp_n_premiers(ch, k) =
   si k=0 alors ch sinon supp_n_premiers(reste(ch), k-1)
```

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

Fin réponse

▷ **Question 12:** *retourne* : $\left\{ \begin{array}{l} Chaine \mapsto Chaine \\ \text{retourne la chaine lue en sens inverse} \end{array} \right.$

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.

retourne(ch) -- version bourinne
```
si ch = chvide alors chvide
               sinon adj(dernier(ch), retourne1(saufdernier(ch)))
```

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 : on prend une pile supplémentaire, on passe le premier de la pile de départ sur l'autre, et on recommence avec la deuxieme de la pile de départ. Si ca aide pas, faut détailler un exemple : A

trier ABC ⇝

ABC	$\emptyset$
BC	A
C	BA
$\emptyset$	CBA

⇝ résulat = CBA

retourne(ch) -- avec helper
```
retourne2(ch):
  retourne2_helper(ch,chvide)

retourne2_helper(ch_todo,ch_done):
  si ch_todo = chvide alors ch_done
                      sinon retourne2_helper(suite(ch_todo),
                                             adj(premier(ch_todo), ch_done))
```

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 ca 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 interresse 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* : $\begin{cases} \textit{Chaine} \times \textit{Chaine} \mapsto \textit{Chaine} \\ \text{retourne les deux chaines concaténées} \end{cases}$

Réponse

version brutale: $O(n^2)$

```
concat1(ch1,ch2):
  si ch1 = chvide alors ch2
                sinon concat1(saufdernier(ch1),
                              adj(dernier(ch1), ch2))
```

¡¡¡¡¡¡¡ HEAD :TD/02-td-recursivite/02-td-recursivite.tex Pour aller plus vite, il faut laisser ch1 à l'envers le temps de travailler au lieu de passer son temps à aller à la fin des chaines. On passe de ======= 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 ¿¿¿¿¿¿¿ 782d82c154b08feb3830f98a7940e2cc8a5e24ce :TD/02-td-recursivite/02-td-recursivite.tex $O(n^2)$ 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'appeller helper
BA	CDEF	récursion dans helper
A	BCDEF	récursion dans helper
∅	ABCFED	Cas terminal de la récursion dans helper

$\leadsto$ résultat = ABCDEF

version avec helper: $O(n)$

```
concat2_helper(ch1,ch2): (un peu mieux)
  si ch1 = chvide alors ch2
                sinon concat2_helper(suite(ch1),
                                     adj(premier(ch1),ch2))

concat2(ch1,ch2):
  concat2_helper(retourne(ch1),ch2)
```

Fin réponse

▷ **Question 14:** *min_ch* : $\begin{cases} \textit{Chaine} \mapsto \textit{caractère} \\ \text{retourne le caractère le plus petit de la chaîne} \end{cases}$

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

Réponse

min_ch(ch)

```
PRECONDITION: ch != chvide
si suite(ch) = chvide alors premier(ch)
                      sinon min(premier(ch), min_ch(suite(ch)))
```

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

Fin réponse

▷ **Question 15:** *croissante* : $\begin{cases} \textit{Chaine} \mapsto \textit{booléen} \\ \text{retourne si la chaine est croissante (dans l'ordre lexicographique)} \end{cases}$

Réponse

croissante(ch)

```
si ch=chvide ou suite(ch)=chvide alors
  VRAI
sinon
```


CORRECTION

```
  si premier(ch) < premier(suite(ch)) alors
    croissante(suite(ch))
  sinon
    FAUX
  finsi
finsi
```

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

Fin réponse

▷ **Question 16:** *nnaturels* : $\begin{cases} \mathbb{N} \mapsto \textit{Chaine} \\ \text{retourne une chaine formée des n premiers entiers naturels} \end{cases}$

Dans un premier temps, on construira $\{n, n-1, n-2, \ldots, 3, 2, 1\}$ avant de construire $\{1, 2, 3, \ldots, n\}$.

Réponse

Version simple qui donne la liste à l'envers

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

Version trichée qui donne la chaine à l'endroit:

```
nnaturels2(n):
  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:

```
nnaturels3(n):
  nnaturels3_helper(1,n-1)

nnaturels3_helper(n, todo):
  si todo = 0 alors chvide
            sinon adj(n, nnaturels3_helper(n+1,todo-1)
```

Fin réponse

▷ **Question 17:** *palindrome* : $\begin{cases} \textit{Chaine} \mapsto \textit{booléen} \\ \text{retourne VRAI si la chaine est un palindrome} \end{cases}$

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)

```
si longueur(ch) <= 1 alors
  VRAI
sinon
  si premier(ch) = dernier(ch) alors
    palindrome(suite(saufdernier(ch)))
  sinon
    si premier(ch) = ' ' alors
      palindrome(suite(ch))
    sinon
      si dernier(ch) = ' ' alors
        palindrome(saufdernier(ch))
      sinon
        FAUX
      finsi
    finsi
  finsi
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

CORRECTION

▷ **Question 18:** $anagramme : \left\{ \begin{array}{l} Chaine \times Chaine \mapsto bool\acute{e}en \\ \text{retourne VRAI si les chaines sont des anagrammes l'une de l'autre} \end{array} \right.$

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

Réponse

anagramme(ch1,ch2)

```
si ch1=chvide et ch2=chvide alors
  VRAI
sinon
  si est_membre(premier(ch1), ch2) alors
    anagramme(suivant(ch1), supprime(premier(ch1), ch2))
  sinon
    FAUX
  finsi
finsi
```

Fin réponse

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

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

Réponse

union(ch1,ch2)

```
si ch1=chvide alors
  si ch2=chvide alors
    chvide
  sinon
    union(ch2,chvide) -- Pour virer les doublons de ch2
  finsi
sinon
  si est_membre(premier(ch1), suite(ch1)) ou est_membre(premier(ch1), ch2) alors
    union(suite(ch1),ch2)
  sinon
    adj(premier(ch1),union(suite(ch1),ch2))
  finsi
finsi
```

Fin réponse

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

Réponse

difference(ch1,ch2)

```
si ch2 = chvide alors
  ch1
sinon
  si ch1 = chvide alors
    chvide
  sinon
    si est_membre(premier(ch1),ch2) alors
      difference(suite(ch1),ch2)
    sinon
      adj(premier(ch1), difference(suite(ch1),ch2))
    finsi
  finsi
finsi
```

Fin réponse