DS INFO N°4

Exercice 1 (Questions de cours).

1. Donner la définition de la complexité amortie d'un algorithme. Quelle est la complexité amortie d'une succession d'invocations de l'opération `push` à partir d'un tableau vide dans le cas d'une implémentation de pile à l'aide d'un tableau redimensionnable ? (pas de démo demandée)
2. On considère un algorithme dont la complexité est définie en fonction d'une taille n de donnée, par la relation de récurrence suivante : $T(n) = 4T\left(\left\lceil \frac{n}{3} \right\rceil\right) + n$. Donner un ordre de grandeur asymptotique de son comportement quand la taille de la donnée d'entrée devient très grande. Justifier par un calcul.
3. Donner 3 implémentations concrètes différentes vues en cours pour la structure de données abstraite de **file** : une avec des maillons chaînés, une avec un tableau de taille fixe, une avec des piles. Faites un dessin et donner quelques éléments clés pour expliquer chaque implémentation.

Corrigé de l'exercice 1.

[Retour à l'énoncé]

1. La complexité amortie est la complexité observée en moyenne lorsque l'on considère des exécutions successives interdépendantes à partir d'une entrée initiale, qui est donc modifiée par chacune des invocations successives. Elle ne dit rien sur une exécution isolée avec une entrée particulière, mais garantit un équilibre, un lissage de la complexité sur une séquence d'opérations enchaînées interdépendantes.
   La complexité amortie de la fonction `push` est constante dans ce cas.

2. $T(n) = 4T\left(\left\lceil \frac{n}{3} \right\rceil\right) + n$: on est dans le cas du théorème maître avec $a = 4$, $b = 3$ et $f(x) = x$, c'est-à-dire polynomiale avec $\beta = 1$. Comme $b^\beta = 3 < a = 4$, on s'attend donc à une complexité en $\Theta\left(n^{\log_3(4)}\right) \approx \Theta\left(n^{1.26}\right)$. Comme le théorème maître n'est pas au programme, on fait un petit calcul pour justifier. On se place pour commencer dans le cas $n = 3^p \Leftrightarrow p = \log_3(n)$. On réinjecte récursivement la relation de récurrence dans elle-même p fois jusqu'à retomber sur le cas d'arrêt de l'algorithme $T(1)$. Cela donne :

$$\begin{aligned}
T(n) &\in \Theta\left(4T\left(3^{p-1}\right) + 3^p\right) \\
&\in \Theta\left(4 \times \left(4T\left(3^{p-2}\right) + 3^{p-1}\right) + 3^p\right) \\
&\in \Theta\left(\ldots\right) \\
&\in \Theta\left(4^p T(1) + \textstyle\sum_{k=0}^{p-1} 4^k 3^{p-k}\right) \\
&\in \Theta\left(4^p T(1) + 3^p \textstyle\sum_{k=0}^{p-1} \left(\frac{4}{3}\right)^k\right) \\
&\in \Theta\left(4^p T(1) + 3^p \left(\frac{4}{3}\right)^p\right) \quad \text{(somme geo. + simplif.}\Theta\text{)} \\
&\in \Theta\left(4^p\right)
\end{aligned}$$

   Or,

$$4^p = 4^{\log_3(n)} = \left(3^{\log_3(4)}\right)^{\log_3(n)} = 3^{\log_3(4)\times\log_3(n)} = 3^{\log_3(n^{\log_3(4)})} = n^{\log_3(4)}$$

On a donc montré que $T(n) \in \Theta\left(n^{\log_3(4)}\right)$ quand n est une puissance de 3, $n = 3^p$. Pour le cas général, il faut encadrer n avec des puissances de 3 et procéder par encadrement de la somme en se rappelant la définition (encadrement) de Θ.

3. On a vu les implémentations suivantes de la structure de données abstraite de file :

Par maillons chaînés : avec deux pointeurs `front` et `last` respectivement sur le premier et le dernier maillon de la file, pour pouvoir insérer un élément un queue de file en temps constant ;

Par un tableau circulaire : la propriété de contiguïté mémoire des tableaux permet d'utiliser un adressage direct en temps constant de n'importe quel élément de la file. On stocke les indices `front` et `last` correspondant respectivement à au premier et au dernier élément de la file. Retirer l'élément de tête de la file revient à incrémenter l'indice `front`. Pour ajouter un élément en queue de file on incrémente `last` mais, pour éviter d'être limité par la taille fixe du tableau, on empile de manière à ce que, si on atteint le bout du tableau, on continue de manière circulaire en début de tableau, ce qui revient à faire un modulo sur l'indice `last`. Faire un dessin (aller voir le dessin, dans le cours de la séquence 8) ;

Par deux piles : on utilise deux piles `back` et `front` : on enfile en empilant sur `back` et on défile en dépilant `front`. Quand `front` est vide, on inverse `back` et on le copie dans `front`. Cette méthode permet des opérations de complexité amortie constante et se prête à une implémentation de file immuable (réalisée en OCaml en TP). Faire un dessin là aussi. (aller voir le dessin, dans le cours de la séquence 8)

Exercice 2.

On propose la fonction OCaml `rev` ci-dessous implémentée avec l'opérateur de concaténation `@`. On note n la taille de la liste donnée en entrée.

```
# let rec rev l =
  match l with
  | [] -> []
  | h::t -> (rev t) @ [h];;
val rev : 'a list -> 'a list = <fun>
```

1. Donner un ordre de grandeur asymptotique de sa complexité, en justifiant proprement par des calculs.

2. Réécrire cette fonction `rev` pour qu'elle soit linéaire en la taille n de la liste donnée en entrée.

Corrigé de l'exercice 2.

[Retour à l'énoncé]

1. On note $T(n)$ la complexité temporelle de l'algorithme où n est la taille de la liste donnée en entrée. On a la relation de récurrence suivante :

$$\begin{aligned} T(0) &= T_0 \\ T(n) &= \underbrace{T(n-1)}_{\text{appel récursif de } \texttt{rev} \text{ sur la queue de liste, de taille } n-1} + (n-1) \end{aligned}$$

Le deuxième terme $(n-1)$ correspond au coût de recomposition, ici, le coût de l'opérateur de concaténation `@`. En OCaml, les listes sont implémentées concrètement

en utilisant une technique de maillons chaînés. Pour pouvoir concaténer deux listes, il faut aller se placer à la fin de la première liste, ce qui nécessite de parcourir tous les éléments de la première liste et entraîne une complexité linéaire en la taille de la première liste. Ici, il faut donc parcourir les $n-1$ éléments de la queue qui a été renversée par l'appel récursif.
On réinjecte proprement la relation de récurrence dans elle-même $n-1$ fois jusqu'à atteindre le cas de base $n=0$ correspondant à la liste vide :

$$\begin{aligned} T(n) &= T(n-1) + (n-1) \\ &= (T(n-2) + (n-2)) + (n-1) \\ &= ((T(n-3) + (n-3)) + (n-2)) + (n-1) \\ &= \dots \\ &= T(0) + \sum_{k=1}^{n} (n-k) \\ &= T_0 + \sum_{k=0}^{n-1} k \text{ (ré-indiçage)} \\ &= T_0 + \frac{n(n-1)}{2} \in \Theta(n^2) \text{ (on a même un équivalent } \sim \frac{n^2}{2} \text{ en fait !)} \end{aligned}$$

L'algorithme a donc une complexité temporelle asymptotique quadratique !

2. On peut s'affranchir de l'utilisation de l'opérateur de concaténation `@` en utilisant une stratégie avec un accumulateur de type `'a list`, initialisé avec la liste vide :

```
# let rev_mieux l =
  let rec aux l acc =
    match l with
    |[] -> acc
    |h::t -> aux t (h::acc)
  in
  aux l [];;
val rev_mieux : 'a list -> 'a list = <fun>
```

Cette fonction a maintenant une complexité linéaire, car le coût de la recomposition (opérateur `::`) est constant. En effet, en OCaml, les listes sont implémentées concrètement en utilisant une technique de maillons chaînés, dans laquelle l'ajout en tête de liste avec l'opérateur `::` est une opération qui s'effectue en temps constant $\Theta(1)$. On a donc :

$$\begin{aligned} T(0) &= T_0 \\ T(n) &= \underbrace{T(n-1)}_{\substack{\text{appel récursif de } \texttt{rev} \text{ sur la queue de liste, de taille } n-1}} + \Theta(1) \end{aligned}$$

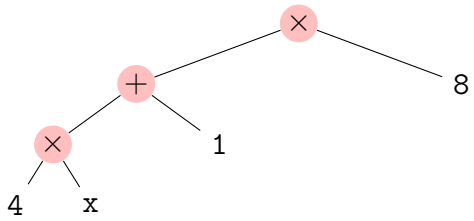
ce qui, par réinjection (très facile), entraîne un coût asymptotique linéaire par rapport à la taille n de la liste donnée en entrée : $T(n) \in \Theta(n)$.

Remarque : cette version nous a permis de gagner en complexité temporelle, mais elle permet aussi de faire d'une pierre deux coups et de gagner en complexité spatiale. En effet, la fonction améliorée est récursive terminale, ce qui permet de réutiliser le même bloc d'activation pour chaque appel récursif et réduit considérablement l'occupation mémoire dans le segment de pile de la mémoire attribuée au processus.

Exercice 3 (Représentation d'une expression algébrique par un arbre binaire).
Une expression algébrique peut être représentée par un arbre dont les nœuds contiennent des lexèmes. Un lexème est :

- soit une valeur numérique : on se limitera aux entiers, par exemple 3, -7, 11 ;
- soit l'indéterminée x, aussi appelée variable (on se limite aux expressions algébriques à une seule indéterminée)
- soit une opération binaire, et on se limitera aux 3 opérations binaires $+$, $-$, $\times$. On ne traite pas le cas des opérations unaires, comme par exemple l'opérateur $-$ dans l'expression (-2) car on considère que le symbole $-$ fait partie de la valeur numérique entière dans ce cas.

Par exemple, l'expression $(4x+1)\times 8$ peut être représentée par l'arbre :



1. Dessiner l'arbre **binaire** (deux fils maximum par nœuds) correspondant à l'expression algébrique
$$(3+2(x+1))\times((x+6)(8x-3))\,,$$
en respectant les parenthèses pour la mise sous forme d'arbre binaire.
2. Expliquez quelles propriétés de l'addition et de la multiplication permettent toujours de se ramener à un arbre **binaire**. Y-a-t-il unicité ?
3. Quelle(s) propriété(s) doit vérifier l'arbre si l'expression algébrique est bien formée ?
4. Donner le résultat affiché par les parcours pré-fixe, infixe et post-fixe pour
   - l'arbre donné en exemple dans l'énoncé
   - l'arbre dessiné à la question 1.

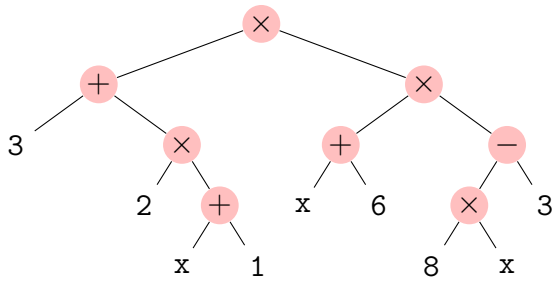
   Quel parcours correspond à notre manière habituelle d'écrire les expressions algébriques ?
5. On définit en OCaml le type `expralg` comme ceci :

```
type 'a expralg = Val of 'a | Var | Add of 'a expralg * 'a expralg |
  Sub of 'a expralg * 'a expralg | Mult of 'a expralg * 'a expralg
```

   a. D'après cette définition, combien y a t-il de manières de construire une expression algébrique ?
   b. Écrire deux lignes de code OCaml permettant de créer les expressions algébriques $(4x+1)\times 8$ et $(3+2(x+1))\times((x+6)(8x-3))$
   c. Écrire une fonction `evaluate: expralg -> int -> int` qui évalue l'expression en une valeur choisie de l'indéterminée x.

Corrigé de l'exercice 3. *[Retour à l'énoncé]*

1. L'arbre binaire correspondant à l'expression algébrique $(3+2(x+1))\times((x+6)(8x-3))$ est :



2. On peut toujours mettre l'expression algébrique sous la forme d'un arbre binaire en utilisant l'associativité de l'addition et de la multiplication, qui nous permet de traiter toujours deux opérandes par deux opérandes :

$$a+b+c=a+(b+c) \quad a\times b\times c=a\times(b\times c)$$

Il n'y a pas unicité car on peut écrire par exemple (c'est pareil pour la multiplication) :

$$a+b+c=a+(b+c) \text{ ou bien } a+b+c=(a+b)+c$$

ce qui aboutit à deux arbres différents. D'un point de vue syntaxique, les deux expressions sont considérées comme différentes, bien qu'elles aient la même sémantique (elles sont totalement équivalentes mathématiquement).

3. Si l'expression algébrique est bien formée, chaque nœud à deux fils.
Les feuilles de l'arbre correspondent soit à des valeurs numériques, soit à l'indéterminée x.
Tous les nœuds internes (nœuds qui ne sont pas des feuilles) correspondent à des opérations algébriques.

4. Pour l'arbre correspondant à l'expression $(4x+1)\times 8$:

Parcours préfixe : $\times + \times 4\, x\, 1\, 8$

Parcours infixe : $4\times x+1\times 8$

Parcours postfixe : $4\, x\times\ 1+\ 8\times$

Pour l'arbre correspondant à l'expression $(3+2(x+1))\times((x+6)(8x-3))$:

Parcours préfixe : $\times + 3\times\ 2+\ x\, 1\times + x\, 6-\times 8\, x\, 3$

Parcours infixe : $3+2\times x\ +\ 1\times\ x+6\times\ 8\times\ x-3$

Parcours postfixe : $3\, 2\ x\ 1+\times\ +x\, 6+8\, x\times\ 3-\times\times$

a. Il y a 5 manière de construire une expression algébrique :

- soit l'expression algébrique se résume à une valeur numérique (constructeur `Val`)
- soit l'expression algébrique se résume à une l'indéterminée (variable) x (constructeur `Var`)
- soit l'expression algébrique a été construite inductivement en sommant deux expressions algébriques existantes (constructeur `Add`)
- soit l'expression algébrique a été construite inductivement en soustrayant deux expressions algébriques existantes (constructeur `Sub`)
- soit l'expression algébrique a été construite inductivement en multipliant deux expressions algébriques existantes (constructeur `Mult`)

b. Voici comment créer les deux arbres correspondant à la création de ces deux expressions algébriques :

```
(* expr1 = (4x+1)*8 *)
let expr1 = Mult(Add(Mult(Val(4),Var),Val(1)),Val(8));;
val expr1 : int expralg = Mult (Add (Mult (Val 4, Var), Val 1), Val 8)
#
(* expr2 = ( 3+2(x+1) ) * ( (x+6) * (8x-3) ) *)
let expr2 = Mult
              (
                Add( Val(3),
                     Mult(Val(2), Add( Var, Val(1)) )
                   ),
                Mult(
                      Add(Var, Val(6)),
                      Sub(
                          Mult(Val(8), Var),
                          Val(3)
                        )
                     )
              );;
val expr2 : int expralg =
  Mult (Add (Val 3, Mult (Val 2, Add (Var, Val 1))),
   Mult (Add (Var, Val 6), Sub (Mult (Val 8, Var), Val 3)))
```

c. Voici une proposition d'implémentation pour la fonction `evaluate`, ainsi que quelques tests d'évaluation sur les deux expressions algébriques créées.

```
# let rec evaluate e x =
  match e with
  | Val(v) -> v
  | Var -> x
  | Add(e1, e2) -> (evaluate e1 x) + (evaluate e2 x)
  | Sub(e1, e2) -> (evaluate e1 x) - (evaluate e2 x)
  | Mult(e1, e2) -> (evaluate e1 x) * (evaluate e2 x);;
val evaluate : int expralg -> int -> int = <fun>
# evaluate expr1 0;;
- : int = 8
# evaluate expr1 1;;
- : int = 40
# evaluate expr2 0;;
- : int = -90
# evaluate expr2 1;;
- : int = 245
```

Exercice 4 (Notation polonaise inversée).

La notation polonaise inverse (aussi appelée RPN pour *reversed polish notation*) correspond à l'écriture post-fixe d'une expression algébrique. Elle a été étudiée et mise en valeur par le mathématicien polonais Lukasiewicz. Elle est utilisée dans certains langages de programmation ainsi que pour certaines calculatrices, notamment celles de la marque Hewlett-Packard. Nous allons voir comment, à l'aide d'une pile, évaluer une expression algébrique donnée en notation polonaise inversée.

On part donc d'une expression algébrique écrite sous sa forme post-fixée. L'idée est d'utiliser une pile pour stocker les opérandes, et d'implémenter les opérations de façon à ce que l'évaluation de l'expression n'utilise que les valeurs au sommet de la pile Par exemple, pour évaluer $(x+4)-2$, noté $x\,4+2-$ en RPN, pour la valeur $x=10$, on effectue les actions suivantes au fur à mesure que l'on dépile la RPN :

- x : on empile 10
- 4 : on empile 4
- $+$: on dépile deux valeurs, que l'on additionne, et on remet le résultat dans la pile.
- 2 : on empile 2
- $-$: on dépile deux valeurs (v_2, puis v_1), que l'on soustrait $v_1 - v_2$, et on remet le résultat dans la pile.

1. Détaillez toutes les étapes de cet algorithme d'évaluation en $x=1$ sur l'expression RPN de l'expression $(4x+1)\times 8$. On écrira les valeurs contenues dans la pile à chaque étape de l'algorithme jusqu'à la fin. On réfléchira à une manière efficace de présenter les étapes de l'algorithme.

2. Quel est le cas d'arrêt de cet algorithme ?

3. Écrire une fonction `rpn : expralg -> string list` qui renvoie une liste de chaînes de caractères correspondant à l'écriture RPN d'une expression algébrique. Par exemple, si l'identificateur `expr3` désigne l'objet de type `expralg` représentant l'expression $(x+4)-2$, l'appel `rpn expr3` renvoie la liste `[''x''; ''4''; ''+''; ''2''; ''-'']`. On pourra utiliser la fonction de conversion `string_of_int` qui convertit un entier en chaîne de caractères. On représentera la multiplication par le caractère `*`.

4. Écrire une fonction `evaluate_rpn: string list -> int -> int` qui évalue une expression algébrique donnée sous sa forme RPN pour un entier x choisi. On pourra utiliser la fonction de conversion `int_of_string` qui convertit une chaîne de caractères en entier, quand cela est possible.

5. Comparer en terme de complexité spatiale la fonction d'évaluation `evaluate` sur les expressions sous forme d'arbres implémentée à l'exercice 2 et la fonction `evaluate_rpn` qui évalue l'expression à partir de son écriture RPN. Quelques phrases percutantes suffisent.

Corrigé de l'exercice 4.

[Retour à l'énoncé]

1. Déroulement de l'algorithme pour l'expression $(4x+1)\times 8$ dont l'écriture RPN est $4\,x\times 1+8\times$, cf exercice précédent, notation postfixe.

Balayage RPN	État de la pile	Opération effectuée
$4\,x\,\times\,1\,+\,8\,\times$	`[]`	On empile 4
$x\,\times\,1\,+\,8\,\times$	`[4]`	On empile la valeur $x = 1$
$\times 1\,+\,8\,\times$	`[1; 4]`	$\times$: on dépile les 2 dernières valeurs, on les multiplie et on empile le résultat
$1\,+\,8\,\times$	`[4]`	On empile 1
$+\,8\,\times$	`[1;4]`	$+$: on dépile les 2 dernières valeurs, on les somme et on empile le résultat
$8\,\times$	`[5]`	On empile 8
$\times$	`[8; 5]`	$\times$: on dépile les 2 dernières valeurs, on les multiplie et on empile le résultat
	`[40]`	RPN entièrement balayée : renvoie la valeur contenue dans la pile et fin

2. L'algorithme s'arrête quand tous les caractères de l'écriture RPN ont été balayés. Il ne doit rester qu'une seule valeur dans la pile. Si ce n'est pas le cas, c'est que l'expression algébrique, et donc sa RPN, sont mal construites. Dans ce cas, on crée un cas d'arrêt d'erreur.

3. Voici une implémentation possible de la fonction `rpn`, avec quelques tests (on doit retrouver les notations postfixe de l'exercice précédent)

```
(* version utilisant l'operateur de concatenation *)
let rec rpn e  =
  match e with
  |Val(v) -> [string_of_int v]
  |Var -> ["x"]
  |Add(e1, e2)  -> ((rpn e1)@(rpn e2))@["+"]
  |Sub(e1, e2)  -> ((rpn e1)@(rpn e2))@["-"]
  |Mult(e1, e2) -> ((rpn e1)@(rpn e2))@["*"];;
val rpn : int expralg -> string list = <fun>
```

On peut proposer une implémentation plus performante sans l'opérateur de concaténation (mais toujours non récursive terminale !)

```
(* version sans l'operateur de concatenation *)
let rpn e =
let rec aux e acc =
  match e with
  |Val(v) -> (string_of_int v)::acc
  |Var -> "x"::acc
  |Add(e1, e2)  -> "+"::(aux e2 (aux e1 acc))
  |Sub(e1, e2)  -> "-"::(aux e2 (aux e1 acc))
  |Mult(e1, e2) -> "*"::(aux e2 (aux e1 acc))
in
List.rev (aux e []);;
val rpn : int expralg -> string list = <fun>
```

4. Voici une implémentation possible de la fonction `evaluate_rpn`

```
# let evaluate_rpn e_rpn valx =
  let rec aux e_rpn valx pile =
  match (e_rpn, pile) with
  | ([], [v]) -> pile (* cas d'arret: balayage rpn fini *)
  | ([], _) -> failwith "Expression algebrique mal construite"
  | (h::t,_) when h = "x"          ->  aux t valx (   valx::pile  )
  | (h::t, v1::v2::q) when h = "+" ->  aux t valx ( (v1+v2)::q )
  | (h::t, v1::v2::q) when h = "-" ->  aux t valx ( (v2-v1)::q ) (* attention ici pour la soustraction, non sym *)
  | (h::t, v1::v2::q) when h = "*" ->  aux t valx ( (v1*v2)::q )
  | (h::t,_)                       ->  aux t valx ( int_of_string(h)::pile  )
  in
  match e_rpn with
  | [] -> failwith "Expression algebrique vide!"
  | _ -> let res = aux e_rpn valx [] in
         match res with
         |h::[] -> h
         |_ -> failwith "Expression algebrique mal construite";;
val evaluate_rpn : string list -> int -> int = <fun>
```

On peut effectuer quelques tests pour valider que l'évaluation de la RPN avec `evaluate_rpn` donne bien le même résultat que l'évaluation de l'expression algébrique de type `expralg` avec `evaluate` :

```
(* compare resultats results deux methodes  *)

(* test 1*)
let e1_rpn = rpn expr1;;
val e1_rpn : string list = ["4"; "x"; "*"; "1"; "+"; "8"; "*"]
# evaluate_rpn e1_rpn 0;;
- : int = 8
# evaluate expr1 0;;
- : int = 8
# evaluate_rpn e1_rpn 1;;
- : int = 40
# evaluate expr1 1;;
- : int = 40
#
(* test 2*)
let e2_rpn = rpn expr2;;
val e2_rpn : string list =
  ["3"; "2"; "x"; "1"; "+"; "*"; "+"; "x"; "6"; "+"; "8"; "x"; "*"; "3"; "-";
   "*"; "*"]
# evaluate_rpn e2_rpn 0;;
- : int = -90
# evaluate expr2 0;;
- : int = -90
```

5. C'est surtout la complexité spatiale qui va changer car la récursivité est terminale et il n'y a pas d'empilement de blocs d'activation liés aux appels récursifs. De plus, dans ce bloc d'activation réutilisé à chaque appel, l'algorithme n'utilise au maximum que deux registres pour stocker les deux valeurs sur lesquelles il va y avoir une opération à faire. L'encombrement spatial est donc bien moindre que pour l'autre fonction, où la récursivité n'est pas terminale (empilement de blocs) et où OCaml recopie les deux sous-arbres dans le tas à chaque appel (bien que le GC les libère dès qu'ils ne sont plus accessibles) !

Exercice 5 (Algorithme d'Euclide et théorème de Lamé).

1. (Question de cours, étudiée au dernier DS) Écrire rapidement une fonction OCaml **récursive** `euclide` implémentant l'algorithme d'Euclide qui calcule le PGCD de deux entiers a et b. On se limitera au cas des entiers naturels, c'est-à-dire que $a \geq 0$ et $b \geq 0$.

2. De quelle(s) quantité(s) dépend a priori la complexité de l'algorithme ?

3. On considère la suite de Fibonacci définie par :

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+2} &= F_{n+1} + F_n \end{aligned}$$

On dit que la suite de Fibonacci est définie par une relation de récurrence d'ordre 2 car on a besoin des deux termes précédents pour calculer le terme courant.

a. Calculer les termes de la suite de Fibonacci jusqu'à F_5.

b. Montrer que la suite de Fibonacci est positive et strictement croissante pour $n \geq 2$.

c. Montrer que la reste de la division euclidienne de F_{n+2} par F_{n+1} est égal à F_n.

4. On admet dans la suite l'ordre de grandeur asymptotique suivant : $F_n \underset{n\to+\infty}{\sim} \frac{1}{\sqrt{5}}\varphi^n$ où $\varphi = \frac{1+\sqrt{5}}{2}$ est le nombre d'or.

a. Expliquer pourquoi on peut utiliser n'importe quelle fonction logarithme $\log_\alpha$, avec un réel α tel que $\alpha > 1$, pour effectuer une analyse asymptotique **en ordre de grandeur** faisant intervenir un logarithme.

b. Montrer que, pour tout $n \geq 1$, si l'algorithme d'Euclide effectue n appels récursifs pour calculer `pgcd(a,b)`, alors cela nous donne une bonne inférieure sur les entrées : $a \geq F_{n+2}$ et $b \geq F_{n+1}$. *Indication : procéder par récurrence simple en écrivant très clairement la propriété démontrée.*

c. On considère que la complexité temporelle de l'algorithme T est correctement représentée par le nombre d'appels récursifs n effectués pour les deux entrées choisies. Déduire des questions précédentes une domination asymptotique de $T(a, b)$ quand a et b deviennent grands.

d. Montrer que, $\forall n \in \mathbb{N}$, l'appel de l'algorithme d'Euclide récursif avec $a \leftarrow F_{n+1}$ et $b \leftarrow F_{n)}$ va engendrer exactement n appels récursifs.

e. En quoi ce dernier résultat permet-il d'affiner l'analyse asymptotique effectuée ?

Ce résultat est connu sous le nom de **théorème de Lamé**.

Corrigé de l'exercice 5.

[Retour à l'énoncé]

1. Voici une implémentation possible récursive de l'algorithme d'Euclide en OCaml :

```
# (* precondition pour cet exo: a et b positifs *)
let rec euclide a b =
  if (b = 0) then a
  else (euclide b (a mod b) );;
val euclide : int -> int -> int = <fun>
```

2. A priori, la complexité temporelle T dépend des deux valeurs d'entrée a et b : $T = T(a, b)$.

3. **a.** $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$

b. Ces deux propriétés se démontrent très simplement par récurrence.

c. On a, d'après la relation de récurrence :

$$F_{n+2} = F_{n+1} + F_n = F_{n+1} \times 1 + F_n \leftrightarrow a = bq + r$$

avec $a = F_{n+2}$, $b = F_{n+1}$, $q = 1$ et $r = F_n$, et il ne faut pas oublier de vérifier que $0 \leq r < b$, ce qui est le cas ici d'après la question précédente car $\forall n \in \mathbb{N}$, $F_n \geq 0$ $F_n < F_{n+1}$ d'après la question précédente. Donc $r = F_{n+2}$ est bien le reste de la division euclidienne de $a = F_{n+2}$ par $b = F_{n+1}$.

4. **a.** Par définition, $\log_\alpha(x) = \dfrac{\ln(x)}{\ln(\alpha)}$. Soit deux valeurs réelles distinctes α et α' strictement supérieures à 1.

$$\log_\alpha(x) = \frac{\ln(x)}{\ln(\alpha)} = \frac{\ln(x)}{\ln(\alpha')} \times \frac{\ln(\alpha')}{\ln(\alpha)} = K\log_{\alpha'}(x)$$

avec $K = \dfrac{\ln(\alpha')}{\ln(\alpha)}$. Ainsi

$$\log_\alpha(x) \underset{x\to+\infty}{\sim} K\log_{\alpha'}(x)$$

Et en particulier, $\log_\alpha(x) \in \Theta(\log_{\alpha'}(x))$ et $\log_{\alpha'}(x) \in \Theta(\log_{\alpha'}(x))$. On peut donc utiliser n'importe quelle fonction logarithme pour les études asymptotiques en ordre de grandeur.

b. On se limite au cas $a > 0$ et $b \geq 0$ deux entiers positifs, tels que $a > b$. On peut le supposer sans perte de généralité car, de toute façon, il sont rangés dans l'ordre dès le deuxième appel de l'algorithme, et le cas $a = b$ est sans intérêt car on s'arrête dès le deuxième appel vu que $a \bmod b = 0$. On écrit proprement la propriété à démontrer :

$\mathcal{P}(n)$: Pour tout a et b tels que $a > b \geq 0$, si l'appel `pgcd(a,b)` effectue n appels récursifs, alors $a \geq F_{n+2}$ et $b \geq F_{n+1}$

Initialisation : S'il y a $n = 0$ appel récursif, on a bien $a \geq 1 = F_2$ et $b \geq 0 = F_1$ avec les hypothèses sur a et b, car $a > b \geq 0$.
S'il y a 1 appel récursif, cela signifie que $b > 0$ (sinon on s'arrêterait directement), donc $b \geq 1 = F_2$ et, par hypothèse $a > b \geq 1$ donc $a \geq 2 = F_3$, la propriété est donc vraie.

Hérédité : On fixe $a > b \geq 0$ et on suppose la propriété vraie au rang n. On suppose maintenant que l'appel `pgcd(a,b)` effectue $(n + 1)$ appels récursifs. D'après l'algorithme, cela signifie que l'appel `pgcd(b, a mod b)` effectue n appels récursifs.
On applique l'hypothèse de récurrence à l'appel récursif `pgcd(b, a mod b)`, donc avec $a \leftarrow b$ et $b \leftarrow a \bmod b$. On en déduit donc que :

$$b \geq F_{n+2} \quad a \bmod b \geq F_{n+1}$$

On a donc déjà $b \geq F_{n+2}$. Il nous reste à montrer $a \geq F_{n+3}$. Or, $a = b \times q + a \bmod b$ avec $q \geq 1$ (sinon, cela contredirait l'hypothèse $a > b$). On a donc :

$$a = b \times q + a \bmod b \underbrace{\geq}_{\text{car } q\geq 1} b + a \bmod b \underbrace{\geq}_{\text{par hyp. rec}} F_{n+2} + F_{n+1} \underbrace{=}_{\text{déf. Fibonacci}} F_{n+3}$$

On a donc montré que $a \geq F_{n+3}$ et $b \geq F_{n+2}$: $\mathcal{P}(n+1)$ est vraie.

Conclusion : Par principe de récurrence, la propriété $\mathcal{P}(n)$ est vraie, $\forall n \geq 0$.

c. On considère que la complexité temporelle de l'algorithme $T(a,b) = n$ où n est le nombre d'appels récursifs effectués pour ces deux entrées. Il s'agit maintenant de trouver un ordre de grandeur asymptotique de T en faisant réapparaître a et b à partir de n. Si $T(a,b) = n$, alors, d'après la question précédente, on sait que $b \geq F_{n+1}$. D'après l'équivalent asymptotique fourni par l'énoncé :

$$F_n \underset{n\to+\infty}{\sim} \frac{1}{\sqrt{5}}\varphi^n$$

Ainsi, pour n'importe quel choix de $\varepsilon > 0$ aussi petit que l'on veut, $\exists n_0 \in \mathbb{N}$ tel que :

$$\forall n \geq n_0,\ F_n \geq (1-\varepsilon)\frac{\varphi}{\sqrt{5}}\varphi^n$$

$$\forall n \geq n_0, \qquad b \geq F_{n+1} \geq (1-\varepsilon)\frac{\varphi^{n+1}}{\sqrt{5}} \Leftrightarrow \log_\varphi(b) \geq n+1- \underbrace{\log_\varphi(\frac{\sqrt{5}}{(1-\varepsilon)})}_{\approx 1.67 \text{ compris entre 1 et 2}} \geq n$$

dont on déduit :

$$n-1 \leq \log_\varphi(b)$$

et on a donc, pour n assez grand, c'est-à-dire b assez grand puisque $b \geq F_{n+1}$ et que la suite de Fibonacci est strictement croissante pour n grand :

$$T(a,b) = n \leq \log_\varphi(b) - 1 < \log_\varphi(b)$$

Ainsi, a tendant lui aussi vers $+\infty$ pour n grand car $a > b$ et on a :

$$T(a,b) \in \underset{a \text{ et } b\to+\infty}{O}(\log_\varphi(b))$$

On a obtenu pour le moment une domination (un grand O).

d. D'après la question préliminaire $\forall n \in \mathbb{N}$, $F_{n+2} \bmod F_{n+1} = F_n$. Ainsi, les appels récursifs successifs de l'algorithme donnent :

$$\texttt{euclide}(F_{n+1}, F_n) = \texttt{euclide}(F_n, F_{n+1} \bmod F_n) = \texttt{euclide}(F_n, F_{n-1}) = \ldots = \texttt{euclide}(\underbrace{F_1}_{=1}, \underbrace{F_0}_{=0}) = 1$$

L'appel de l'algorithme d'Euclide récursif avec $a \leftarrow F_{n+1}$ et $b \leftarrow F_n$ va donc engendrer exactement n appels récursifs.

e. L'énoncé nous dit que $F_n \underset{n\to+\infty}{\sim} \frac{1}{\sqrt{5}}\varphi^n$. Donc, dans le cas de l'appel $\texttt{euclide}(F_{n+1}, F_n)$, la complexité, correspondant au nombre d'appels récursifs effectués, est :

$$n \underset{n\to+\infty}{\sim} \log_\varphi(F_n) + \log_\varphi(\sqrt{5}) \underset{n\to+\infty}{\sim} \log_\varphi(F_n)$$

.

Ainsi, pour l'appel euclide(a, b) avec $a \leftarrow F_{n+1}$ et $b \leftarrow F_n$,

$$T(a,b) \underset{a,b\to+\infty}{\sim} \log_\varphi(b)$$

On a trouvé un cas d'appel pour lequel la complexité au pire donné par le grand O de la question précédente est atteint. On peut donc affirmer que cette domination est, en fait, un ordre de grandeur et on a :

$$T(a,b) \in \underset{a \text{ et } b\to+\infty}{\Theta}(\log_\varphi(b))$$

f. Complément. Le théorème de Lamé est parfois énoncé comme ceci : le nombre d'itérations/d'appels récursifs de l'algorithme d'Euclide `euclide a b` est majoré par le nombre de chiffres de l'écriture décimale de b, multiplié par 5. Voyons d'où vient ce résultat.

Nous avons montré que, pour b assez grand :

$$T(a,b) \leq \log_\varphi(b)$$

Le nombre de chiffres de l'écriture décimale de b est $\lceil \log_{10}(b) \rceil$ (cf TD de début d'année). Faisons apparaître cette valeur dans la majoration précédente :

$$T(a,b) \leq \log_\varphi(b) \leq \frac{\ln(10)}{\ln(\varphi)} \log_{10}(b) \leq \frac{\ln(10)}{\ln(\varphi)} \lceil \log_{10}(b) \rceil$$

Et, avec la calculatrice, on trouve :

$$K = \frac{\ln(10)}{\ln(\varphi)} \approx 4.78 < 5$$

On a donc montré que :

$$T(a,b) < 5 \times \lceil \log_{10}(b) \rceil$$

Exercice 6 (DM (maths/info) Démonstration de l'ordre asymptotique sur la suite de Fibonacci).

1. Justifier l'égalité matricielle suivante :

$$\begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

2. On définit la suite de vecteurs $(U_n)_{n\in\mathbb{N}}$ de terme général :

$$U_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

Écrire une relation de récurrence définissant la suite $(U_n)_{n\in\mathbb{N}}$, en n'oubliant pas l'initialisation de la récurrence. Quelle est l'ordre de cette récurrence ?

3. On note dans toute la suite :

$$\varphi^+ = \varphi = \frac{1+\sqrt{5}}{2} \quad \text{et} \quad \varphi^- = \frac{1-\sqrt{5}}{2}$$

Montrer que φ^+ et φ^- sont les racines du polynômes $X^2 - X - 1$. En déduire **sans calcul** les valeurs de $\varphi^+ \times \varphi^-$ et $\varphi^+ + \varphi^-$.

4. On note $A \in \mathbb{R}^{2\times 2}$ la matrice à coefficients réels à deux lignes et deux colonnes définie par :

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Montrer l'égalité matricielle suivante :

$$A = PDP^{-1} \text{ où } P^{-1} = \begin{pmatrix} 1 & 1 \\ \varphi^+ & \varphi^- \end{pmatrix}, \quad D = \begin{pmatrix} \varphi^+ & 0 \\ 0 & \varphi^- \end{pmatrix} \text{ et } P = \frac{1}{\varphi^- - \varphi^+} \begin{pmatrix} \varphi^- & -1 \\ -\varphi^+ & 1 \end{pmatrix}$$

Vérifier que P^{-1} est bien la matrice inverse de P.

5. On définit une suite auxiliaire $(V_n)_{n\in\mathbb{N}}$ à partir de la suite $(U_n)_{n\in\mathbb{N}}$ par la relation : $V_n = P^{-1}U_n$. Écrire la relation de récurrence vérifiée par $(V_n)_{n\in\mathbb{N}}$.

6. Donner une formule **vectorielle explicite** pour la suite $(V_n)_{n\in\mathbb{N}}$.

7. En déduire une formule **vectorielle explicite** pour la suite $(U_n)_{n\in\mathbb{N}}$

8. En déduire un ordre de grandeur asymptotique de la suite $(F_n)_{n\in\mathbb{N}}$ quand $n \to +\infty$. *Indication : on s'interrogera sur l'expression précédente, en se demandant quelles quantités dépendent de la variable n et en oubliant pas que les constantes devant les différents termes sont sans intérêt pour les études asymptotiques en ordre de grandeur.*

La démonstration que vous venez d'effectuer cache en fait une théorie mathématique extrêmement utile : la théorie des valeurs propres. Vous la découvrirez l'an prochain et l'utiliserez abondamment, aussi bien en mathématiques qu'en physique.
A retenir : pour résoudre une récurrence sur une suite numérique d'ordre strictement supérieur à 1, je transforme le problème en une récurrence sur une suite vectorielle d'ordre 1... Cela marche aussi pour les équations différentielles !