DS INFO N°3
Corrigé

Exercice 1 (Questions de cours - 10 minutes).

1. Quelle est la complexité **spatiale** de l'algorithme des tours de Hanoï ? Donner une justification en quelques lignes et/ou avec un petit dessin, sans y passer trop de temps. (3 minutes)
2. Écrire en OCaml l'algorithme d'exponentiation rapide dans sa version **itérative** (5 minutes)
3. Donner (sans justification) les **complexités temporelles asymptotiques** des algorithmes suivants, vus en cours ou en TD/TP. Vous préciserez de quoi dépend cette complexité et expliquerez les notations choisies.(1 minute)

Algorithme	Complexité temporelle au pire
Recherche dichotomique	
Exponentiation naïve	
Tours de Hanoï	
Tri à bulles	
Exponentiation rapide	
Tri insertion	
Recherche séquentielle	
Multiplication russe	
Algorithme de recherche naïf de motif dans un texte	
Concaténation de deux listes	

Corrigé de l'exercice 1.

[Retour à l'énoncé]

1. La complexité spatiale de l'algorithme des tours de Hanoï est linéaire. C'est surtout l'empilement des blocs d'activation liés aux appels récursifs qui va faire augmenter l'espace mémoire alloué dans la pile. Comme chaque appel récursif engendre 2 nouveaux appels, on pourrait croire que la complexité spatiale est exponentielle, en $O(2^n)$ mais il n'en est rien car on dépile aussi régulièrement les blocs d'activation lorsqu'un appel atteint son cas de base. Une représentation des appels sous forme d'arbre permet bien de voir qu'il y a 2^n appels, mais que le nombre de blocs d'activations maximal qui pourront être empilés correspond à la profondeur de l'arbre.
2. L'algorithme d'exponentiation rapide s'écrit de manière itérative :

```
# let rec exp_rapide x n =

  let r = ref 1.0 in
  let k = ref n in
  let a = ref x in

  while (!k > 0)
  do
    if (!k mod 2 = 1) then
      r := !r *. !a;
    a := !a *. !a;

    k := !k / 2
  done;
  !r;;
val exp_rapide : float -> int -> float = <fun>
# exp_rapide 2.0 (-1);;
- : float = 1.
# exp_rapide 2.0 3;;
- : float = 8.
# exp_rapide (-1.5) 4;;
- : float = 5.0625
```

3.

Algorithme	Complexité temporelle au pire
Recherche dichotomique	$O(\log_2(n))$ où n est la taille du tableau où est recherchée la valeur
Exponentiation naïve	$O(n)$ où n est la valeur de l'exposant entier
Tours de Hanoï	$O(2^n)$ où n est le nombre de disques à déplacer
Tri à bulles	$O(n^2)$ où n est la taille du tableau à trier
Exponentiation rapide	$O(\log_2(n))$ où n est la valeur de l'exposant entier
Tri insertion	$O(n^2)$ où n est la taille du tableau à trier
Recherche séquentielle	$O(n)$ où n est la taille du tableau où est recherchée la valeur
Multiplication russe	$O(\log_2(x))$ où x est l'un des facteurs (entier) de la multiplication
Algo. de recherche naïf motif	$O(m \times n)$ où m et n tailles motif et texte
Concaténation de deux listes	$O(n_1)$ où n_1 est la taille de la première liste

Exercice 2 (Division euclidienne et algorithme d'Euclide).

1. On définit le *quotient* q et le *reste* r de la division euclidienne de $a \in \mathbb{Z}$ par $b \in \mathbb{N}^*$ (donc $b > 0$) comme l'unique couple d'entiers $q \in \mathbb{Z}$ et $r \in \mathbb{Z}$ vérifiant $a = bq + r$ avec $0 \leq |r| < b$.

Pour cette question, on se limitera au cas où $a \geq 0$.

a. **Pour cette question seulement, la multiplication et la division sont interdites.** Écrire une fonction **itérative** en langage C

```
void div_euclid(int a, int b, int *q, int *r)
```

qui reçoit deux paramètres entiers `a` et `b` et qui calcule le quotient et le reste de la division euclidienne de a par b . On respectera scrupuleusement le prototype imposé pour cette fonction. On pourra faire quelques exemples sur la copie pour se donner des idées et ensuite généraliser.

b. Prouver la terminaison de votre fonction.

c. Prouver la correction de votre fonction.

d. Quelle est la complexité temporelle de cette fonction ?

2. Le *plus grand diviseur commun* de deux entiers a et $b \geq 0$, noté $\text{pgcd}(a, b)$, est le plus grand entier **positif** qui divise à la fois a et b.

a. Que vaut $\text{pgcd}(a, 0)$? Justifier en quelques mots.

b. Montrer que

$$\forall m \in \mathbb{Z}, n \in \mathbb{N},\ \text{pgcd}(m,\ n) = \text{pgcd}(n,\ r) \tag{1}$$

où r est le reste de la division euclidienne de m par n.

c. Utiliser cette propriété de manière répétée pour calculer le PGCD de $a = 63$ et $b = 11$ en décrivant toutes les étapes du calcul.

d. En vous aidant de cet exemple, écrire en **C** un algorithme **itératif**

```
void euclide(int a, int b, int *d)
```

permettant de calculer le PGCD de deux nombres a et b.

e. L'algorithme fonctionne-t-il pour $a < b$? Que se passe-t-il précisément ?

f. Prouver rigoureusement la correction partielle de votre algorithme. *Indication :* on pourra s'appuyer sur le résultat prouvé précédemment (Equation 1).

g. Montrer la *correction totale* de votre fonction.

h. Écrire en OCaml une version **récursive** de l'algorithme d'Euclide.

Corrigé de l'exercice 2.

[Retour à l'énoncé]

1. a. On suppose d'abord que $a \geq 0$ et on fait comme le font les enfants : on compte combien on peut faire rentrer de b dans a. On part de a : on enlève un b à a et tant que le résultat reste positif, on incrémente le compteur q qui compte le nombre de fois où on réussit à rentrer b dans a. La boucle s'arrête donc quand on ne peut plus rentrer de a dans ce qui nous reste.

```c
/**
Fonction div_euclid

Calcule le quotient et le reste de la div euclidienne
de a par b>0
a = bq+r où 0 <= |r| < b
@param  a: nombre entier
@param  b: nombre entier, précondition b > 0
@return q quotient de la div euclidienne de a par b
@return r: reste de la div euclidienne de a par b
*/
// le prototype de l'enonce nous impose de retourner
// les valeurs de q et de r en utilisant un passage par adresse
// attention, division et multiplication interdite
void div_euclid(int a, int b, int *q, int *r)
{
  assert(b > 0);

  bool inversion = false;

  if (a < 0)
  {
    a = a -a -a; // a <- a - 2a  ==> revient à faire inverser a
    inversion = true;
  }

  *q = 0;
  *r = a;

  while (*r >= b)
  {
    // invariant de boucle r = a - b*q
    *r = *r - b;
    *q = *q + 1;
  }

  //complexite O( E(a/b) )

  if (inversion == true)
  {
    *q = *q -(*q) -(*q); // q <- q - 2q ==> revient à faire inverser q
    *r = *r -(*r) -(*r);  // r <- r - 2r ==> revient à faire inverser r
  }

  return;
}
```

Pour gérer le cas $a < 0$, on inverse simplement a avant de faire le même algorithme, et il faut ensuite inverser le quotient et le reste trouvés car

$$a = bq + r \Leftrightarrow -a = b \times (-q) + (-r)$$

b. On utilise comme variant la suite $(r_k)_{k\in\mathbb{N}}$ définie par :

$$\begin{aligned} r_0 &= |a| \\ r_k &= r_{k-1} - b \end{aligned}$$

r_k désigne la suite des valeurs successives stockées dans la variable r.

Une récurrence immédiate montre que la suite $(r_k)_{k\in\mathbb{N}}$ est à valeurs entières car a et b le sont. De plus, il s'agit d'une suite arithmétique de raison $-b$, qui est donc strictement négative car on a supposé que $b > 0$. La suite est donc strictement décroissante.

Comme la suite ne fait que décroître strictement, il existe donc nécessairement un indice k_0 à partir duquel la suite devient strictement inférieur à b :

$$\exists k_0 \in \mathbb{N},\ r_{k_0} < b$$

Ainsi, le test de boucle $r_{k_0} \geq b$ devient donc faux et la boucle prend fin. L'algorithme se termine donc quelque soient les entrées a et b vérifiant les préconditions énoncées.

c. L'algorithme est itératif : nous allons donc tenter une preuve à l'aide d'un invariant de boucle. On se restreint au cas $a \geq 0$ car dans le cas où a est négatif, on effectue simplement un prétraitement et un post traitement supplémentaires qui ont été justifiés mathématiquement lors de l'écriture de l'algorithme. On considère l'invariant de boucle suivant :

$$\mathcal{P} : r = a - q \times b$$

et r, a, b et q désigne les valeurs contenues dans les variables `r`, `a`, `b` et `q`.

Initialisation : Au tout début du tout premier tour de boucle, d'après l'algorithme de la question 1, on a $r = |a|$ et $q = 0$. On vérifie bien que $a = a - 0 \times q$ donc $\mathcal{P}$ est vraie.

Conservation : On note :

— r, a, b et q les valeurs contenues dans les variables `r`, `a`, `b` et `q` au tout début d'un tour de boucle.

— r', a', b' et q' les valeurs contenues dans les variables `r`, `a`, `b` et `q` à la toute fin d'un tour de boucle

On suppose que la propriété est vraie au début d'un tour de boucle quelconque, c'est-à-dire que l'on a $r = a - q \times b$.

D'après l'algorithme, $r' = r - b$ et $q' = q + 1$.

`a` et `b` ne sont jamais modifiées, donc $a' = a$ et $b = b'$.

Ainsi :

$$r' = r - b = a - q \times b - b = a - (q + 1) \times b = a' - q' \times b'$$

Ainsi, $\mathcal{P}$ est vérifiée à la fin du tour de boucle.

Conclusion : Une fois sortis du tout dernier tour de boucle, $\mathcal{P}$ est donc toujours vérifiée. On a donc :

$$r = a - q \times b \Leftrightarrow a = b \times q + r$$

Mais de plus, comme la boucle s'est arrêtée, cela signifie que le test $r \geq b$ n'a pas été validé, et on est donc certains que $r < b$. On a donc bien trouvé q et r tels que :

$$a = bq + r, \text{ avec } 0 \leq r < b\,,$$

ce qui valide la correction de l'algorithme. On a donc prouver la correction totale de l'algorithme.

d. La complexité temporelle de cette fonction dépend des deux paramètres d'entrée a et b. Elle est proportionnelle au nombre d'itérations, c'est-à-dire au quotient euclidien $\lfloor \frac{a}{b} \rfloor$. La complexité temporelle est donc en $\Theta(\lfloor \frac{a}{b} \rfloor)$ (grand theta).

2. a. 0 est divisible par n'importe quel entier. Par ailleurs, le plus entier positif qui divise m est $|m|$. On en déduit que le plus grand entier positif qui divise à la fois m et n est $|m|$:

$$\text{pgdc}(m,\, 0) = |m|$$

b. Notons $d = \text{pgdc}(m,\, n)$ et $d' = \text{pgdc}(n, r)$. On cherche donc à montrer que $d = d'$. On note $x|y$ pour indique qu'un entier $x > 0$ divise un entier y.

Montrons que $d|d'$. Par définition du PGCD, $d|m$ et $d|n$, ce qui permet d'écrire $m = m'd$ et $n = n'd$.

Par définition du reste de la division euclidienne de a par b, on a : $r = m - nq$. Donc $r = m'd - n'dq = d(m' - n'q)$, ce qui montre que $d|r$.

Ainsi d est un diviseur commun à n et à r : on en déduit que $d|\text{pgcd}(n, r)$ et donc $d|d'$.

Montrons maintenant que $d'|d$. Comme $d' = \text{pgdc}(n, r)$, par définition du PGCD, $d'|n$ et $d'|r$.

On note $b = n'd'$ et $r = r'd'$, que l'on injecte dans $m = nq + r$:

$$m = nq + r \Leftrightarrow m = n'd'q + r'd' = d'(n'q + r')$$

On voit donc que $d'|m$.

Nous avons donc $d'|m$ et $d'|n$. Par définition du PGCD, $d'|\text{pgdc}(m, n)$, c'est-à-dire $d'|d$.

Conclusion : on a montré que $d|d'$ et $d'|d$, ce qui signifie que $d = \pm d'$. Mais comme d et d' sont tous les deux des PGCD, on sait qu'ils sont tous les deux positifs par définition du PGCD et donc que :

$$\text{pgdc}(m,\ n) = \text{pgdc}(n, r)$$

c.

Itération			a	b	r	
0	63	$= 11 \times 5 + 8$	63	11	8	$\text{pgcd}(63, 11)$
1	11	$= 8 \times 1 + 3$	11	8	3	$= \text{pgcd}(11, 8)$
2	8	$= 3 \times 2 + 2$	8	3	2	$= \text{pgcd}(8, 3)$
3	3	$= 2 \times 1 + 1$	3	2	$\boxed{1}$	$= \text{pgcd}(3, 2)$
4	2	$= 2 \times 1 + 0$	2	1	0	$= \text{pgcd}(2, 1)$
5	1	$= x \times 0 + 1$	$\boxed{1}$	0	1	$= \text{pgcd}(1, 0) = \boxed{1}$

On en déduit que le PGCD de $a = 63$ et $b = 11$ est donc 1... ce qui est vrai car 11 est premier !

d. Voici une implémentation itérative en C de l'algorithme d'Euclide

```c
/**
Fonction euclide

Calcule le PGCD de deux nombres entiers

@param  a: nombre entier
@param  b: nombre entier, b > 0 (précondition)
@return PGCD de a et de b (nombre entier)
*/
int euclide(int a, int b)
{
  assert(b >= 0);

  int d  = a;
  int ds = b;
  int r;

  while (ds != 0) // Variant: suite (d, d_s)_k,
  {               // k est le num d'itération

    // invariant de boucle
    // P: pgcd(a,b) = pgcd(d, ds)
    r = d % ds;
    d = ds;
    ds = r;
  }

  return d;
}
```

Dans le code, l'évolution de la variable `d` correspond à la colonne a du tableau de suivi de variables, et la variable `ds` correspond à la colonne b.

e. Pour $a < b$, le première étape donne :

$$a = b \times 0 + a$$

et à la deuxième étape, le diviseur b devient dividende et le reste a devient diviseur, on effectuera donc la division euclidienne suivante :

$$b = a \times q + r\,.$$

Tout se passe donc comme si la première étape de l'algorithme échangeait les valeurs d'entrée a et b dans le cas où $a < b$. Ainsi, notre algorithme est naturellement résilient au cas $a < b$.

f. Pour prouver la correction de l'algorithme, on choisit naturellement comme invariant de boucle la propriété :

$$\mathcal{P} : \quad \text{pgdc}(a,\, b) = \text{pgdc}(d, d_s)$$

où d et d_s désignent respectivement les valeurs contenues dans les variables `d` et `ds`.

g. Pour montrer la correction totale, il nous reste à prouver la terminaison de l'algorithme. Pour cela, on introduit le variant $(d_k)_{k\in\kappa}$:

$$\begin{cases} d_0 & = |a| \\ d_1 & = b \\ d_{k+2} = d_k \% d_{k+1} & \end{cases}$$

où on a noté $m\%n$ le reste de la division euclidienne d'un entier m par un entier $n > 0$.

$(d_k)_{k\in\kappa}$ est bien une suite à valeurs entières. Elle est même à valeurs entières positives car on a pris soin de prendre la valeur absolue de a, et on sait que $b > 0$ par hypothèse.

Nous allons montrer par récurrence que la suite est décroissante en montrant la propriété

$$\mathcal{P}(k) : \quad d_{k+1} \leq d_k,\ k \in \mathbb{N}$$

Cas de base : Si $|a| > b$, on a immédiatement que $d_1 = b \leq d_0 = |a|$ et donc $\mathcal{P}(0)$ est vraie.

Si $|a| < b$, $\mathcal{P}(0)$ est fausse mais nous avons vu que, dès la deuxième itération, $d_1 = b$ et $d_2 = a$ et donc $d_2 \leq d_1$ ce qui montre que $\mathcal{P}(1)$ est vraie. Dans ce cas, la propriété sera montrée $\forall k \geq 1$ mais pas pour $k = 0$, mais cela n'a aucune importance pour la démonstration de la terminaison.

Hérédité : Soit $k \geq 1$ et supposons $\mathcal{P}(k)$ vraie, c'est-à-dire $d_{k+1} \leq d_k$.

Alors d_{k+2} est le reste de la division euclidienne de d_k par d_{k+1} :

$$d_k = d_{k+1} \times q + d_{k+2}$$

avec $d_{k+2} < d_{k+1}$. Ce qui prouve $\mathcal{P}(k+1)$

Conclusion : La suite $(d_k)_{k\in\mathbb{N}}$ est décroissante

En fait, il existe deux cas :

— Si la suite $(d_k)_{k\in\mathbb{N}}$ est strictement décroissante, alors est à valeurs entières positives et strictement décroissante : elle est donc stationnaire en 0 à partir d'un certain rang :

$$\exists k_0 \in \mathbb{N},\ \forall k \geq k_0,\ d_k = 0$$

Ainsi, le test de boucle devient faux au bout d'un nombre fini d'itérations et l'algorithme termine.

— Sinon, cela signifie qu'il existe au moins une valeur k' telle que $d_{k'+1} = d_{k'}$. Or, si cela se produit, $d_{k'+2}$ sera nul car le reste de la division euclidienne de $d_{k'}$ par $d_{k'+1}$ est nul : $d_{k'} = d_{k'+1} \times 1 + 0$ Donc le test de boucle deviendra faux et l'algorithme se terminera là encore.

h. Pour prouver la correction de l'algorithme, nous allons utiliser l' invariant de boucle suivant :

$$\mathcal{P} : \quad \text{pgcd}(d, d_s) = \text{pgcd}(a, b)$$

où a, b, d et d_s désignent les valeurs associées aux variables `a`, `b`, `d` et `d_s`.

Initialisation : Au tout début du tout premier tour de boucle $d = a$, $d_s = b$ et la propriété est trivialement vérifiée.

Conservation : Supposons la propriété vérifiée au début d'un tour de boucle.

On note d et d_s les valeurs associées aux variables `d` et `d_s` en début de tour.

On note d' et d'_s les valeurs associées aux variables `d` et `d_s` en fin de tour.

On suppose que la propriété est vraie en début de tour :

$$\text{pgcd}(d, d_s) = \text{pgcd}(a, b)$$

D'après l'algorithme :

$$\begin{aligned} d &= d_s q + r \\ d' &= d_s \\ d'_s &= r \end{aligned}$$

$$\text{pgcd}(d', d'_s) = \text{pgcd}(d_s, r) = \text{pgcd}(d, d_s)$$

et pour la dernière égalité, on a appliqué le résultat de la question 1.2 en remplaçant $m \leftarrow d$ et $n \leftarrow d_s$. En utilisant l'hypothèse de validité de la propriété en début de tour, on a donc :

$$\text{pgcd}(d', d'_s) = \text{pgcd}(d, d_s) = \text{pgcd}(a, b)$$

et le propriété est donc également vérifiée en fin de tour.

Conclusion : En sortie du dernier tour, $d'_s = 0$ (test d'arrêt) donc, d'après la question 1.1 :

$$\text{pgcd}(d', d'_s) = \text{pgcd}(d', 0) = d'$$

Par ailleurs la propriété est vraie pour les dernières valeurs calculées :

$$\text{pgcd}(d', d'_s) = \text{pgcd}(a, b)$$

On a donc :

$$d' = \text{pgcd}(a, b)$$

Comme d' est bien la valeur renvoyée par l'algorithme, on a donc démontré que l'algorithme retourne bien le PGCD de a et de b.

i. On peut directement implémenter l'algorithme de manière récursive en utilisant l'égalité de l'Equation1 :

```
(* euclide récursif
   @param a entier
   @param b entier
   @return pgcd(a, b) *)
let rec euclide a b =
  match b with
  | 0 -> abs(a) (* valeur absolue d'un entier *)
  | _ -> euclide b (a mod b)
;;


euclide 25 10;;
euclide 63 11;;
euclide 63 9;;
euclide (-25) 10;; (* cas a < 0 géré, pgcd positif retourné *)
euclide 10 25;; (* résilient au cas a < b *)
euclide 61 (-9);; (* et même résilient au cas b < 0 en fait :-) *)
```

Exercice 3 (Algorithme de calcul de racine).

Soit a et b deux nombres réels tels que $a < b$ et $f : [a, b] \to \mathbb{R}$ une fonction continue strictement monotone telle que $f(a)$ et $f(b)$ sont de signes opposés.

1. Montrer qu'une telle fonction admet une **unique racine** : $\exists! x_0 \in [a, b]$, $f(x_0) = 0$. *Indication : On pourra notamment faire appel à un théorème du cours de mathématiques.*
2. Soit α et β deux réels non nuls. Quel test (un seul test) permet de répondre à la question : « α et β sont-ils de signes opposés ? »
3. Soit c la valeur centrale de l'intervalle $[a, b]$ et $f(c)$ son image par f. En quoi la connaissance de la valeur $f(c)$ permet d'avancer dans la recherche de la racine ?
4. A partir de la constatation de la question précédente, écrire en OCaml un algorithme `racine` permettant de calculer la valeur de la racine d'une fonction f respectant les critères ci-dessus à ε près. La fonction valeur absolue est incluse dans la bibliothèque standard OCaml et se nomme `abs_float`.
5. Expliquer pourquoi la fonction suivante vérifie bien les préconditions de votre algorithme :
$$\begin{array}{rccc} f : & [0.25, 0.8] & \to & \mathbb{R} \\ & x & \to & \cos(x\pi) \end{array}$$
Quelle est la réponse attendue si l'on donne cette fonction en entrée de l'algorithme ?
6. A l'aide de la calculatrice, tester pas à pas à la main votre algorithme sur la fonction précédente avec $\epsilon = 10^{-2}$, en laissant une trace de votre test, par exemple en faisant un tableau de suivi de variables.
7. Prouver la terminaison de l'algorithme. *Indication : on sort légèrement du cadre mathématique vu dans le cours mais cela ne pose pas de problème particulier ici. Faites vous confiance, la logique mathématique reste la même.*
8. De quels paramètres dépend la complexité de l'algorithme ? Donnez une expression asymptotique de la complexité dans le pire des cas.

Corrigé de l'exercice 3.

[Retour à l'énoncé]

1. Le théorème des valeurs intermédiaires s'applique directement, toutes ses hypothèses sont vérifiées. Cela garantit l'existence d'au moins une racine.

   Il nous reste à prouver l'unicité. Nous allons utiliser pour cela un raisonnement par l'absurde.

   Supposons qu'il existe au moins deux racines x_0 et x_1.

   Il existe au moins un réel y dans $[x_0, x_1]$ qui n'est pas une racine car sinon, cela signifierait que f est constante égale à 0 sur cet intervalle, ce qui contredirait la stricte monotonie.

   On a donc deux possibilités :

   — Si $f(y) > 0$, alors $f(y) - f(x_0) = f(y) - 0 = f(y) > 0$ et $f(x_1) - f(y) = 0 - f(y) = -f(y) < 0$ ce qui contredit la monotonie de f ;

   — De même, si $f(y) < 0$, alors $f(y) - f(x_0) = f(y) - 0 = f(y) < 0$ et $f(x_1) - f(y) = 0 - f(y) = -f(y) > 0$ ce qui contredit encore la monotonie de f.

Dans tous les cas, on aboutit à une contradiction, ce qui prouve l'unicité de la racine de f.

2. Le test permettant de répondre à cette question est $f(\alpha) \times f(\beta) < 0$

3. **Si $f(c)$ et $f(a)$ sont de même signe,** alors c est situé avant la racine : il faut donc chercher la racine dans le sous-intervalle $[c, b]$.

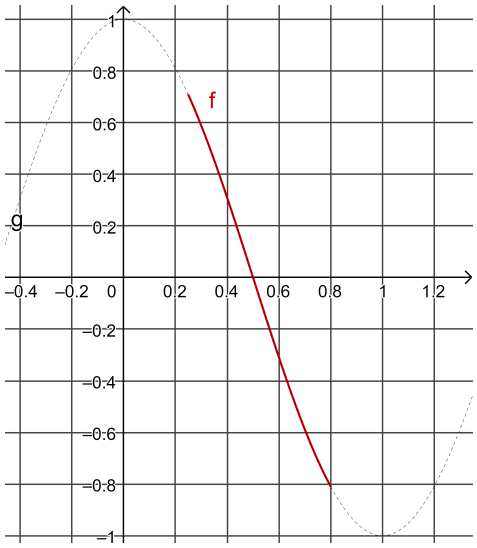
Si $f(c)$ et $f(a)$ sont de signe opposé, , alors c est situé après la racine : il faut donc chercher la racine dans le sous-intervalle $[a, c]$.

4. Nous proposons ici une approche récursive mais une approche itérative est également tout à fait possible :

```
# (* remarque: racine est une fonction d'ordre superieur car l'un
   de ses arguments d'entrée est une fonction *)
let rec racine f a b eps = (* eps tolerance sur la precision de la racine*)
  match (a, b) with
  | (a, b) when (b -. a < eps) ->
     let c = (a +. b) /. 2. in
     Printf.printf "Intervalle [%f,%f], x0=%f f(x0)=%f\n" a b c (f c);
     (* affichage printf facultatif *)
     c

  | _ -> let c = (a +. b) /. 2. in
         let fa = (f a) and fc = f c in
         Printf.printf "Intervalle [%f,%f], c=%f, f(c)=%f\n" a b c fc;
         if (fc *. fa < 0.) then
           racine f a c eps
         else
           racine f c b eps;;
val racine : (float -> float) -> float -> float -> float -> float = <fun>
# let pi = 2.*.acos(0.);;
val pi : float = 3.14159265358979312
# let f x = cos ( x *. pi);;
val f : float -> float = <fun>
# racine f 0.25 0.8 1.e-2;;
```

5. On peut donner l'allure de la fonction f :



— La fonction f proposée est bien continue car cos l'est.

— Elle est strictement décroissante car il s'agit de la composition d'un fonction linéaire strictement croissante (coefficient directeur π strictement positif) et de la fonction cos strictement décroissante sur cet intervalle .

— $f(0.25) = \dfrac{\sqrt{2}}{2} > 0$ et $f(0.8) = \cos(0.8\pi) < \cos(0.5\pi) = 0$ (et on a utilisé le fait que cette fonction est décroissante).
Ainsi, $f(0.25)$ et $f(0.8)$ sont bien de signes opposés.

Le fonction f proposée vérifie donc bien toutes les préconditions.

6. On sait que, dans l'intervalle $[0.25, 0.8]$, la seule valeur annulant le cosinus est $\frac{\pi}{2}$. Donc on s'attend à trouver une valeur approchée de $x_0 = 0.5$ pour la racine de f.

7. Effectuons un tableau de suivi de variables :

Intervalle $[a, b]$	c	$f(c)$
[0.250000, 0.800000]	0.525000	−0.078459
[0.250000, 0.525000]	0.387500	0.346117
[0.387500, 0.525000]	0.456250	0.137012
[0.456250, 0.525000]	0.490625	0.029448
[0.490625, 0.525000]	0.507812	−0.024541
[0.490625, 0.507812]	0.499219	−0.002454
[0.499216, 0.507812]	0.503516	−0.011044

L'algorithme s'arrête à la dernière étape car l'amplitude de l'intervalle [0.499216, 0.507812] est bien d'amplitude strictement inférieure à 10^{-2}. La valeur retournée est $x_0 \approx 0.503516$.

Attention, la précision est à calculer sur la racine x_0 et non sur l'image $f(x_0)$. Dans cet exemple, on voit bien que la racine est bien à 10^{-2} car elle est incluse dans l'intervalle [0.499216, 0.507812] d'amplitude strictement inférieure à 10^{-2}. Par contre, on a $|f(x_0)| = 0.011044 > 10^{-2}$.

8. On note a_k et b_k les paramètres a et b donnés en entrée du k-ième appel récursif. D'après l'algorithme, on a les relations de récurrence suivantes :

$$a_0 = a$$
$$a_{k+1} = \begin{cases} a_k & \text{si} f(c) \text{ et } f(a) \text{ sont de signes opposés} \\ c_k & \text{sinon} \end{cases}$$

$$b_0 = b$$
$$a_{k+1} = \begin{cases} c_k & \text{si} f(c) \text{ et } f(a) \text{ sont de signes opposés} \\ b_k & \text{sinon} \end{cases}$$

et $c_k = \dfrac{a_k + b_k}{2}$.

Pour prouver la terminaison, on utilise la suite $(l_k)_{k\in\mathbb{N}}$, de terme général $l_k = b_k - a_k$, qui représente l'amplitude de l'intervalle $[a_k, b_k]$ donné en paramètre d'entrée du k-ième appel récursif.

Ici, $(l_k)_{k\in\mathbb{N}}$ n'est pas à valeurs entières mais on va pouvoir s'en sortir malgré tout.

$$\begin{cases} l_0 = b - a \\ l_{k+1} = \begin{cases} c_k - a_k & \text{si} f(c) \text{ et } f(a) \text{ sont de signes opposés} \\ b_k - c_k & \text{sinon} \end{cases} \end{cases}$$

Comme $c_k = \dfrac{a_k + b_k}{2}$ (cette fois-ci, c'est une vraie division réelle !), on a en fait $c_k - a_k = b_k - c_k = \dfrac{b_k - a_k}{2}$ et on obtient une expression très simple de la suite $(l_k)_{k\in\mathbb{N}}$:

$$\begin{cases} l_0 = b - a \\ l_{k+1} = \dfrac{l_k}{2} \end{cases}$$

Un récurrence évidente permet de montrer que $(l_k)_{k\in\mathbb{N}}$ est à valeurs strictement positives car $l_0 = b - a > 0$ par hypothèse : $\forall k,\ l_k > 0$

$(l_k)_{k\in\mathbb{N}}$ est une suite géométrique de raison $\dfrac{1}{2} < 1$: elle est donc strictement décroissante.

Cette suite est donc décroissante et minorée par 0 : elle converge.

Un passage à la limite dans la relation de récurrence montre que sa seule limite possible est 0. Donc $(l_k)_{k\in\mathbb{N}}$ tend vers 0.

Par définition, pour ε fixé, il existe un rang k_0 à partir duquel $l_k < \varepsilon$. Pour cet appel récursif k_0, le cas de base s'activera donc et l'imbrication des appels récursifs prendra fin. Ceci prouve la terminaison de l'algorithme.

9. La complexité de l'algorithme dépend des 3 paramètres a, b et ε. Il ne faut pas oublier ce dernier : en effet, plus la précision recherchée sera importante, plus l'algorithme sera coûteux.

On peut donner une formule explicite pour la suite $(l_k)_{k\in\mathbb{N}}$:

$$l_k = \frac{b-a}{2^k}$$

L'algorithme s'arrête quand :

$$\begin{aligned}
& l_k < \varepsilon \\
\Leftrightarrow\ & \frac{b-a}{2^k} < \varepsilon \\
\Leftrightarrow\ & \frac{b-a}{\varepsilon} < 2^k \\
\Leftrightarrow\ & \log_2\left(\frac{b-a}{\varepsilon}\right) < k \\
\Leftrightarrow\ & k = \lfloor \log_2\left(\frac{b-a}{\varepsilon}\right) \rfloor + 1
\end{aligned}$$

L'algorithme effectue donc $\lfloor \log_2\left(\dfrac{b-a}{\varepsilon}\right) \rfloor + 1$ appels. La complexité est donc en $\Theta\left(\lfloor \log_2\left(\dfrac{b-a}{\varepsilon}\right) \rfloor\right)$