Devoir Surveillé 2

16 novembre 2024

INFORMATIQUE MP2I

DURÉE DE L'ÉPREUVE : **4 heures**

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Ce sujet comporte huit pages numérotées de 1/19 à 19/19

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Tournez la page S.V.P.

Vue d'ensemble du sujet

Ce sujet est composé de 7 parties indépendantes, utilisant les langage de programmation C et OCaml.

Les différentes parties sont indépendantes et peuvent être traités dans n'importe quel ordre. Il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie.

- La partie I s'intéresse aux chaînes de caractères et à la création et l'affichage de motifs en C et en OCaml.
- La partie II s'intéresse aux nombres parfaits, puis au nombres premiers, en C.
- La partie III s'intéresse à l'analyse d'une implémentation en OCaml de l'algorithme d'Euclide étendu.
- La partie IV s'intéresse à la modification de chaînes de caractères, en C, et aux choix d'un jeu de tests.
- La partie V s'intéresse à un problème de dénombrement d'objets satisfaisant une définition par récurrence, en OCaml.
- La partie VI s'intéresse à différents systèmes de numérotations, autre que la base b classique, en OCaml.
- La partie VII s'intéresse à l'analyse d'un programme permettant d'énumérer les anagrammes d'une chaîne de caractère, en OCaml.

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. En langage C, il est inutile de rappeler que les entêtes `<assert.h>`, `<stdbool.h>`, etc. doivent être inclus.

Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle.

Partie I. Chaînes de caractères et affichage

Pyramide (C – 5 pts)

On souhaite créer une fonction en C qui prend en entrée un entier positif n et qui affiche un motif pyramidal sur n lignes, comme dans l'exemple ci-dessous, à droite. On propose la fonction ci-dessous, à gauche, où on suppose que la fonction `void print_char(char c)` affiche un caractère.

affichage de pyramide — C

```c
void print_pyramide(int n) {
  for (int i = 0; i < n; i = i + 1) {
    for (int j = 0; j <= i; j = j + 1) {
      print_char('#');
    }
    print_char('\n');
  }
}
```

```
#
##
###
####
#####
######
#######
```

☐ 1 – Combien de caractères sont affichés par cette fonction ?

Réponse 1. Il suffit de compter le nombre de fois que la fonction `print_char` est appelée.
Pour chaque itération de la boucle extérieure, la boucle intérieure effectue $i+1$ itérations. On a donc le nombre de caractères affichés égal à :

$$\sum_{i=0}^{n-1} i+2 = \sum_{i=2}^{n+1} i = \frac{n(n+3)}{2}$$

On souhaite désormais écrire cette chaîne de caractère dans une variable `char pyramide[?]` plutôt que de l'afficher sur la sortie standard.

☐ 2 – Par quel nombre doit-on remplacer ? entre les crochets si on veut une pyramide de taille 7 (celle représentée en exemple) ?

Réponse 2. Il y a $\frac{n(n+3)}{2} = \frac{7\times10}{2} = 35$ caractères, il faut donc un tableau de taille 36 caractères car il faut une case pour stocker le caractère sentinel `'\0'`.

☐ 3 – Écrire la fonction `void pyramide(int n, char pyramide[])` qui prend en entrée un entier positif n et un tableau de taille suffisante, et qui le rempli avec la chaîne de caractères souhaitée.

Réponse 3.

C

```c
void pyramide(int n, char pyramide[]) {
  int pos = 0;
  for (int i = 0; i < n; i = i + 1) {
    for (int j = 0; j <= i; j = j + 1) {
      pyramide[pos] = '#';
      pos = pos + 1;
    }
    pyramide[pos] = '\n';
    pos = pos + 1;
  }
  pyramide[pos] = '\0';
}
```

Motifs (OCaml – 5 pts)

☐ 4 – Écrire une fonction `mult : string -> int -> string` qui prend en entrée une chaîne de caractère s et un entier n et qui renvoie la chaîne contenant n fois s d'affilé. On ne demande pas que cette fonction soit le plus efficace possible.

Réponse 4. Version naïve :

```
let rec mult str n =
  if n <= 0 then ""
  else str ^ mult str (n-1)
```

Version exponentiation rapide/square and multiply :

```
let rec mult str n =
  if n <= 0 then ""
  else
    let half = mult str (n/2) in
    if n mod 2 = 0 then half ^ half
    else str ^ half ^ half
```

☐ 5 – Écrire une fonction `motif : int -> unit` qui prend en entrée un entier n et qui dessine le motif constitué d'un carré de croisillons # de taille n privé d'un carré en diagonal qui est constitué d'espaces. Voici les premiers motifs à renvoyer pour n allant de 1 à 9 :

```
#  ## ###  ####  #####  ######  #######  ########  #########
   ## # #  #  #  ## ##  ##  ##  ### ###  ###  ###  #### ####
      ###  #  #  #   #  #    #  ##   ##  ##    ##  ###   ###
           ####  ## ##  #    #  #     #  #      #  ##     ##
                 #####  ##  ##  ##   ##  #      #  #       #
                        ######  ### ###  ##    ##  ##     ##
                                #######  ###  ###  ###   ###
                                         ########  #### ####
                                                   #########
```

Réponse 5.

```
let motif n =
  let rec ligne i =
    if i = 0 then mult "#" n
    else
      let nb_hash = (n+1)/2-i in
      let hashes = mult "#" nb_hash in
      hashes ^ (mult " " (n - 2 * nb_hash)) ^ hashes
  in
  let rec boucle i =
    let l = ligne i in
    if i < n / 2 then
      begin
        print_endline l;
        boucle (i+1);
        print_endline l;
      end
    else if n mod 2 = 1 then
      print_endline l
  in boucle 0
```

Partie II. Diviseurs, nombres parfaits et nombres premiers (C – 15 pts)

Un nombre parfait est un nombre dont la somme des diviseurs propre est égale à lui même.

$$n \text{ est parfait } \Leftrightarrow \sum_{d\in\mathcal{D}} d = n, \mathcal{D} = \{d \mid d \text{ divise } n \text{ et } d \neq n\}$$

☐ 6 – Écrire une fonction `bool divise(int d, int n)` qui prend en entrée deux entiers positifs d et n et qui renvoie un booléen valant `true` si et seulement si d divise n.

Réponse 6.

```c
bool divise(int d, int n) {
  return (0 == n % d);
}
```

☐ 7 – Écrire une fonction `bool est_parfait(int n)` qui prend en entrée un entier positif n et qui renvoie un booléen valant `true` si et seulement si n est un nombre parfait.

Réponse 7.

```c
bool est_parfait(int n) {
  int acc = 0;
  for (int d = 1; d < n; d = d + 1) {
    if (divise(d,n)) {
      acc = acc + d;
    }
  }
  return (acc == n);
}
```

On veut maintenant implémenter le cribble d'Erathostène pour trouveer tous les nombres premiers inférieurs à un certain nombre n. On rappelle le principe de cet algorithme :

— on commence avec un tableau contenant tous les nombres jusque n ;
— on barre 0 et 1 ;
— on parcours le tableau et chaque fois qu'on tombe sur un nombre qui n'est pas barré : ce nombre est un nombre premier, on barre tous ses multiples plus grand que lui.

On représentera ce tableau par un tableau de booléens `bool cribble[]` dont la case `bool[i]` vaut `true` si `i` n'est pas barré et `false` si `i` est barré.

☐ 8 – Écrire une fonction `void init_cribble(int n, bool cribble[])` qui prend en argument un entier positif n et un tableau de booléens de taille n et qui rempli le tableau en accord avec les deux premières étapes du cribble d'Erathostène.

Réponse 8.

C
```c
void init_cribble(int n, bool cribble[]) {
  cribble[0] = false;
  cribble[1] = false;
  for (int i = 2; i < n; i = i + 1) {
    cribble[i] = true;
  }
}
```

☐ 9 – Étant donné deux entiers positifs i et n, combien y-a-t-il de multiples m de i tels que $i < m \leq n$?

Réponse 9. Si $n = q \times i + r$, avec $0 \leq r < i$, alors les multiples de i à prendre en compte sont $1 \times i, 2 \times i, 3 \times i, \ldots, q \times i$, il y a donc $q = n/i$ (au sens de la division entière) multiples de i inférieurs à n.

☐ 10 – Écrire une fonction `void barre_multiples(int n, bool cribble[], int i)` qui prend en argument un entier positif n, un tableau de booléens de taille n, et un entier positif i et qui assigne la valeur `false` à tous les multiples de i compris entre i exclu et n inclus.

Réponse 10.

C
```c
void barre_multiples(int n, bool cribble[], int i) {
  for (int j = 2 * i; j < n; j = j + i) {
    cribble[j] = false;
  }
}
```

☐ 11 – Écrire une fonction `void erathostene(int n, bool cribble[])` qui prend en argument un entier positif n et un tableau de booléens de taille n et qui assigne la valeur `false` à tous les nombres composites (i.e. qui ne sont pas premiers).

Réponse 11.

```c
void erathostene(int n, bool cribble[]) {
  init_cribble(n, cribble);
  for (int i = 0; i < n; i = i + 1) {
    if (cribble[i]) {
      barre_multiple(in, cribble, i);
    }
  }
}
```

☐ 12 – Écrire la fonction `int main(int argc, char *argv[])` d'un programme qui affiche tous les nombres premiers jusque 1 000 en utilisant le cribble d'Erathostène.

Réponse 12.

```c
int main(int argc, char *argv[]) {
  int cribble[1000];
  erathostene(1000, cribble);
  for (int i = 0; i < n; i = i + 1) {
    if (cribble[i]) {
      printf("%d\n",i);
    }
  }
  return 0;
}
```

Partie III. Algorithme d'Euclide étendu (OCaml – 10 pts)

L'algorithme d'Euclide permet de calculer rapidement de pgcd de deux nombres en prenant à chaque itération le reste par la division euclidienne. L'algorithme d'Euclide étendu est une adaptation permettant d'obtenir aussi les coefficients de Bézout du couple d'entiers :

```
val extended_euclid : int -> int -> int * int * int = <fun>
```

qui à partir d'un couple d'entiers positifs (a, b) renvoie (d, u, v) où d est le pgcd de a et b, et $d = au + bv$.
Voici une implémentation de cet algorithme en OCaml :

```
let extended_euclid a b =
  let rec euclid_aux a b u v u' v' =
    if b = 0 then
      (a, u, v)
    else
      let (q, r) = (a / b, a mod b) in
      euclid_aux b r u' v' (u - q * u') (v - q * v')
  in
  euclid_aux a b 1 0 0 1
```

Pour analyser cette fonction, notons (a_n) et (b_n) les suites des arguments `a` et `b` lors des appels récursifs à `euclid_aux`. De même, en remarquant que `u'` est la prochaine valeur prise par `u`, on note (u_n) et (v_n) les suites des arguments `u` et `v`, et on a `u'` qui vaut u_{n+1} et `v'` qui vaut v_{n+1}.

☐ 13 – Montrer que durant la suite d'appels récursifs à `euclid_aux`, $(2b_n + a_n)$ est une suite d'entiers strictement décroissante. Que peut-on en déduire ?

Réponse 13. On montre par récurrence que (a_n) et (b_n) sont des suites d'entiers positifs. On suppose que la précondition a et b sont des entiers positifs est vérifiée. (Initialisation)
Par définition de la division euclidienne, $r = a \mod b$ est positif, d'où l'hérédité.

Il reste à montrer que $2r + b < 2b + a$, avec $r = a \mod b$ pour montrer la décroissance stricte :
Par définition de la division euclidienne, $0 \leq r < b$ et $r \leq a$.

La suite $(2b_n + a_n)$ est une suite d'entiers positifs strictement décroissante, elle est nécessairement finie, et la fonction `euclid_aux` termine pour toutes valeurs de a et b positives.

☐ 14 – Montrer que pour tout n, on a $a_n = a_0 u_n + b_0 v_n$.

Réponse 14. Notons (q_n, r_n) le quotient et le reste de la division euclidienne de a_n par b_n. Initialisation : On a $u_0 = 1$ et $v_0 = 0$ d'où l'égalité au rang $n = 0$.
On a $u_1 = 0$, $v_1 = 1$ et $a_1 = b_0$ d'où l'égalité au rang $n = 1$. Hérédité : On a, $u_{n+2} = u_n - q_n u_{n+1}$ et $v_{n+2} = v_n - q_n v_{n+1}$. Si $a_n = a_0 u_n + b_0 v_n$ et $a_{n+1} = b_n = a_0 + u_{n+1} + b_0 v_{n+1}$, alors :

$$\begin{aligned} a_0 u_{n+2} + b_0 v_{n+2} &= a_0 u_n - a_0 q_n u_{n+1} + b_0 v_n - b_0 q_n v_{n+1} \\ &= a_n - q_n b_n \\ &= r_n \\ &= b_{n+1} \\ &= a_{n+2} \end{aligned}$$

☐ 15 – Montrer que pour tout n, le pgcd de a_n et b_n est le pgcd de a_0 et b_0.

Réponse 15. Il suffit de montrer que a_{n+1} et b_{n+1} ont même pgcd que a_n et b_n pour conclure par récurrence, c'est-à-dire que pour a et b entiers positifs, a et b ont même pgcd que b et $r = a \mod b$.

$\exists k \in \mathbb{N}, r = a - kb$
Soit d le pgcd de a et b, alors d divise $a - kb = r$.
Soit d' le pgcd de b et r, alors d' divise $r + kb = a$.
d divise d' et d' divise d, ils sont donc égaux.

☐ 16 – Déduire des deux questions précédentes la correction de la fonction `extended_euclid`.

Réponse 16. La valeur renvoyée par la fonction `extended_euclid` est le triplet (a_n, u_n, v_n) du dernier appel récursif. On sait que $b_n = 0$ dans ce cas, et le pgcd de a_n et b_n (et donc de a_0 et b_0 par la question précédente) est donc a_n. De plus, u_n et v_n vérifient l'identité de Bézout par la question précédent la précédente.
On en conclue donc que la fonction est correcte.

☐ 17 – Utiliser la fonction précédente pour définir une fonction

```
inv_mod : int -> int -> int = <fun>
```

qui prend en entrée un nombre premier p et un entier n et qui renvoie l'inverse de n modulo p, c'est-à-dire le nombre entier x compris entre 0 et $p-1$ tel que $xn \mod p = 1$.

Réponse 17. Si p et n sont premiers entre eux, alors leur pgcd est 1 et l'identité de Bézout devient $1 = up + vn$, alors vn est congru à 1 modulo p, et v est l'inverse de n modulo p.

```
let inv_mod p n =
  if n mod p = 0 then raise Division_by_zero
  else let (d, u, v) = extended_euclid p n in
  assert (d = 1);
  let inv = v mod p in (* on veut x entre 0 et p-1 *)
  (inv + p) mod p (* au cas où inv serait négatif *)
```

Partie IV. leet speak (C – 10 pts)

Le 1337 ou leet speak consiste à remplacer certaines lettres par des chiffres ou autres caractères ASCII. Nous considérons la variante suivante :

— 'l' et 'L' sont transformés en '1';
— 'e' et 'E' sont transformés en '3';
— 't' et 'T' sont transformés en '7'.

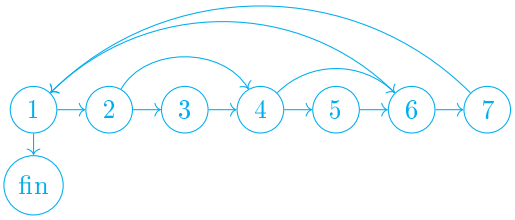
☐ 18 – Écrire une fonction `void leet(char mot[])` qui prend en entrée une chaîne de caractères et qui lui applique le code 1337.

Réponse 18.

```c
void leet(char mot[]) {
  for (int i = 0; mot[i] != '\0'; i = i + 1) { // 1
    if ('l' == mot[i] || 'L' == mot[i]) { // 2
      mot[i] = '1'; // 3
    }
    if ('e' == mot[i] || 'E' == mot[i]) { // 4
      mot[i] = '3'; // 5
    }
    if ('t' == mot[i] || 'T' == mot[i]) { // 6
      mot[i] = '7'; // 7
    }
  }
  // fin
}
```

☐ 19 – Déterminer le graphe de flot de contrôle de cette fonction.

Réponse 19.



☐ 20 – Donner le chemin dans le graphe correspondant à l'exécution de la fonction sur l'entrée "Hello".

Réponse 20. (1) – (2) – (4) – (6) – (1) – (2) – (4) – (5) – (6) – (1) – (2) – (3) – (4) – (6) – (1) – (2) – (3) – (4) – (6) – (1) – (2) – (4) – (6) – (1) – (fin)

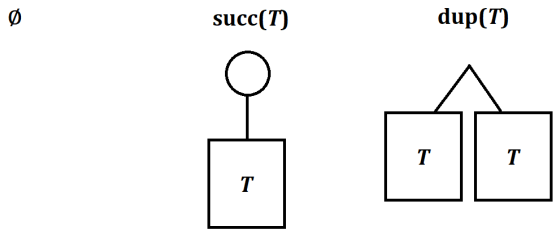
☐ 21 – Proposer un jeu de tests réalisant une couverture des branches.

Réponse 21. Avec le test `"Hello"` qui devient `"H311o"`, il manque les branches (6) – (7) et (7) – (1) à couvrir. On peut utiliser à la place par exemple le test `"LETleto"` qui devient `"137137o"` pour tester à la fois toutes les branches et toutes les façons possibles de vérifier les conditions.

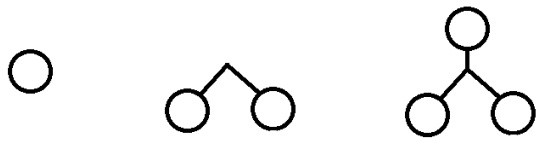
Partie V. Dénombrement : arbres binaires symétriques (OCaml – 10 pts)

On se propose de compter le nombre d'arbres binaires symétriques que l'on peut construire de la manière suivante. Un arbre binaire symétrique peut être soit :

- Un arbre vide $\emptyset$, ne contenant aucun élément ;
- Un arbre $\mathbf{succ}(\boldsymbol{T})$ contenant un élément, puis un sous-arbre binaire symétrique $\boldsymbol{T}$;
- Un arbre $\mathbf{dup}(\boldsymbol{T})$ contenant deux copies d'un sous-arbre binaire symétrique $\boldsymbol{T}$ non vide.



Par exemple, on a représenté ci-dessous les arbres $\mathbf{succ}(\emptyset)$, $\mathbf{dup}(\mathbf{succ}(\emptyset))$, et $\mathbf{succ}(\mathbf{dup}(\mathbf{succ}(\emptyset)))$ qui possèdent respectivement 1, 2, et 3 éléments :

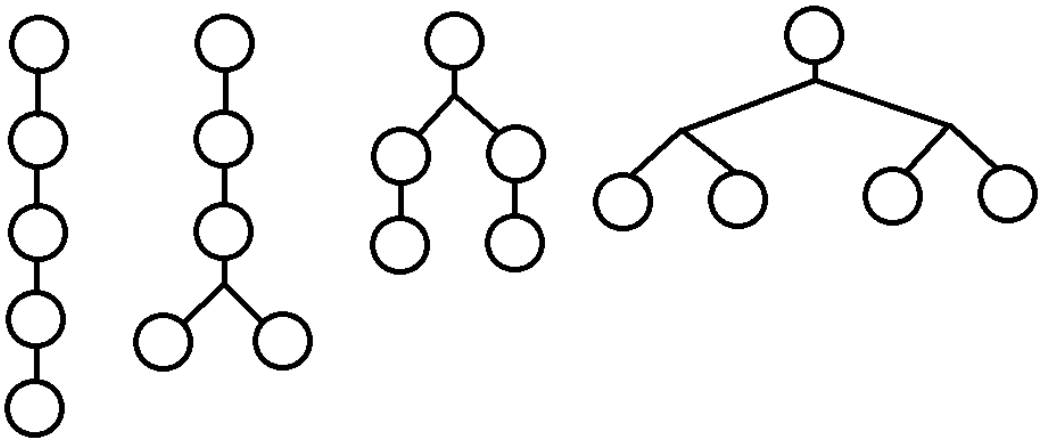


☐ 22 – Si un arbre $\boldsymbol{T}$ contient k éléments, combien d'éléments comptent $\mathbf{succ}(\boldsymbol{T})$ et $\mathbf{dup}(\boldsymbol{T})$?

Réponse 22. $\mathbf{succ}(\boldsymbol{T})$ contient $\boldsymbol{T}$ et un élément donc $k+1$ éléments.
$\mathbf{dup}(\boldsymbol{T})$ contient 2 copies de $\boldsymbol{T}$ donc $2k$ éléments.

☐ 23 – Dessiner tous les arbres symétriques contenant 5 éléments.

Réponse 23. Il y a 5 arbres symétriques contenant 5 éléments :



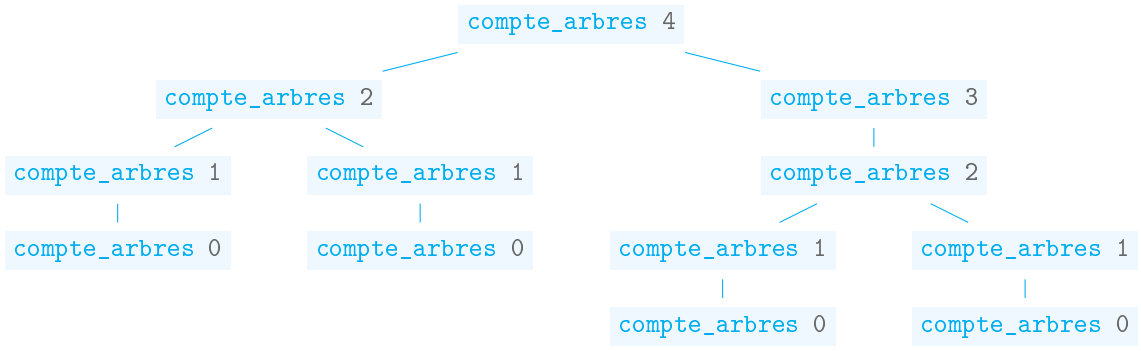
☐ 24 – Écrire une fonction récursive `compte_arbres : int -> int` qui prend en entrée un nombre n et qui calcule le nombre d'arbres symétriques différents de taille n.

Réponse 24.

```
let rec compte_arbres n =
  assert (n >= 0);
  if n = 0 then
    1
  else if n mod 2 = 0 then
    compte_arbres (n/2) + compte_arbres (n-1)
  else
    compte_arbres (n-1)
```

☐ 25 – Dessiner l'arbre des appels récursifs pour l'expression `compte_arbres 4`.

Réponse 25. Le voici :



Partie VI. Changeons de base (OCaml)

Rappel : Écrire un nombre n en base $b \geq 2$, c'est l'écrire $\overline{n_\ell n_{\ell-1} \dots n_1 n_0}^b$,
avec $\forall i \in [\![0, \ell]\!]$, $n_i \in [\![0, b-1]\!]$, tels que $n = \sum_{i=0}^{\ell} n_i b^i$. Par exemple, $13 = \overline{1101}^2$ car $13 = 8+4+1 = 2^3+2^2+2^0$.
Pour trouver l'écriture d'un nombre dans une base b, on procède par divisions euclidiennes successive par b :

$$\begin{aligned} 13 &= 6 \times 2 + 1 \\ 6 &= 3 \times 2 + 0 \\ 3 &= 1 \times 2 + 1 \\ 1 &= 0 \times 2 + 1 \end{aligned}$$

On s'arrête Lorsque le quotient vaut 0, et l'écriture du nombre se lit de bas en haut.

Bases négatives (17 pts)

On peut étendre ce concept pour écrire des nombres en base $-b$, $b \geq 2$, toujours avec des chiffres entre 0 et $b-1$.
Par exemple, en base -10, le nombre 123 s'écrit $\overline{283}^{-10}$ car $123 = 2 \times 100 + 8 \times (-10) + 3 \times 1$.
Et le nombre -123 s'écrit $\overline{1937}^{-10}$ car $-123 = 1 \times (-1000) + 9 \times 100 + 3 \times (-10) + 7 \times 1$.
Les conversions peuvent toujours s'effectuer par divisions euclidiennes successives, mais il faut l'adapter pour les diviseurs négatifs.

☐ 26 – Convertir 42 en base -3.

Réponse 26.

$$\begin{aligned} 42 &= -14 \times (-3) + 0 \\ -14 &= 5 \times (-3) + 1 \\ 5 &= -1 \times (-3) + 2 \\ -1 &= 1 \times (-3) + 2 \\ 1 &= 0 \times (-3) + 1 \end{aligned}$$

Donc $42 = \overline{12210}^{-3}$.
Vérification : $81 + 2 \times (-27) + 2 \times 9 - 3 = 81 - 54 + 18 - 3 = 42$.

☐ 27 – Convertir en décimales $\overline{111111}^{-2}$.

Réponse 27.

$$\begin{aligned} \overline{111111}^{-2} &= (-2)^5 + (-2)^4 + (-2)^3 + (-2)^2 + (-2)^1 + (-2)^0 \\ &= -32 + 16 - 8 + 4 - 2 + 1 \\ &= -21 \end{aligned}$$

Pour implémenter cette conversion en OCaml, nous avons besoin d'une division euclidienne, `/` et `mod` ne conviennent pas lorsque le dividende est négatif :

```
(* On souhaiterait -5 = 2 * (-3) + 1 *)
# (-5) / (-3);;
- : int = 1
# (-5) mod -3;;
- : int = -2
```

☐ 28 – Écrire une fonction `div_e : int -> int -> int * int` qui prend en entrée deux entiers a et b et qui renvoie un couple d'entiers (q, r) tels que :

$$a = q \times b + r, \qquad 0 \leq r < |b|$$

Réponse 28.

```
let div_e a b =
  assert (b <> 0);
  let (q, r) = (a / b,a mod b) in
  if r >= 0 then
    (q, r)
  else if b > 0 then
    (q - 1, b + r)
  else
    (q + 1, r - b)
```

☐ 29 – Proposer un jeu de tests (entrées et sorties correspondantes attendues) que vous utiliseriez pour vérifier votre fonction.

Réponse 29. On cherche à couvrir les différents cas positif/négatif, ainsi que la division par 0.

Jeu de tests pour div_e — REPL

```
# div_e 11 3;;
- : int * int = (3, 2)
# div_e (-11) 3;;
- : int * int = (-4, 1)
# div_e (-11) (-3);;
- : int * int = (4, 1)
# div_e 11 (-3);;
- : int * int = (-3, 2)
# div_e 11 0;;
Exception: Assert_failure ("//toplevel//", 2, 2).
# div_e 0 0;;
Exception: Assert_failure ("//toplevel//", 2, 2).
# div_e 0 3;;
- : int * int = (0, 0)
```

Pour les questions suivantes, les représentations des nombres dans des bases autres que 10 seront de type `string`. Nous utiliserons les opérateurs `^+` et `+^` définis ci-dessous pour ajouter un caractère à droite et à gauche de chaînes de caractères.

Concaténation d'un char et une string — REPL

```
# let (^+) str c = str ^ String.make 1 c;;
val ( ^+ ) : string -> char -> string = <fun>
# let (+^) c str = String.make 1 c ^ str;;
val ( +^ ) : char -> string -> string = <fun>
# ("Bonjour"^+'n')^+'o';;
- : string = "Bonjourno"
# 'a'+^('b'+^"cd");;
- : string = "abcd"
```

☐ 30 – Écrire une fonction récursive `convertis : int -> int -> string` qui prend en entrée un nombre entier n et un nombre entier $b \in \{-10, \dots, -2, 2, \dots, 10\}$, et qui renvoie la représentation de n en base b sous forme d'une string.
On n'oubliera pas de vérifier la précondition sur b à l'aide d'une assertion.

Réponse 30.

```
let to_digit d = char_of_int (d + int_of_char '0')

let rec convertis n b =
  if n = 0 then "0" else
    let (q,r) = div_e n b in
    if q = 0 then
      string_of_int r
    else
      convertis q b ^+ to_digit r
```

Intéressons nous à l'addition.

☐ 31 – Calculer $\overline{27}^{-10} + \overline{13}^{-10}$.

Réponse 31.

$$\begin{aligned}\overline{27}^{-10} + \overline{13}^{-10} &= 2 \times (-10) + 7 + 1 \times (-10) + 3 \\ &= 3 \times (-10) + 10 \\ &= 2 \times (-10) \\ &= \overline{20}^{-10}\end{aligned}$$

☐ 32 – Calculer $\overline{7}^{-10} + \overline{3}^{-10}$.

Réponse 32.

$$\begin{aligned}\overline{7}^{-10} + \overline{3}^{-10} &= 7 + 3 \\ &= 10 \\ &= 100 + 9 \times (-10) \\ &= \overline{190}^{-10}\end{aligned}$$

☐ 33 – Proposer un algorithme permettant d'effectuer l'addition de deux nombres en base -2.
On ne demande pas nécessairement de code dans cette question.

Réponse 33. On pourra s'aider d'une fonction récursive auxilliaire de type

```
string -> string -> char -> string
```

qui prend en argument deux représentations de nombres en base -2 et une retenue '+', '-' ou '=', et qui ajoute les deux nombres en ajoutant 1 si la retenue est +, en retirant 1 si la retenue est '-', et sans rien ajouter ni retirer si la retenue est '='.

```
let ajoute a b =
  let rec ajoute_aux a b r =
    if b = "" && r = '=' then
      a
    else if b = "" then
      ajoute_aux a (retenue r) '='
    else if a = "" then
      ajoute_aux b (retenue r) '='
    else
      let n = String.length a in
      let m = String.length b in
      let (an, ap) = (a.[n-1], String.sub a 0 (n-1)) in
      let (bm, bp) = (b.[m-1], String.sub b 0 (m-1)) in
      if an = '1' && bm = '1' then
        if r = '+' then
          ajoute_aux ap bp '-' ^+ '1'
        else if r = '-' then
          ajoute_aux ap bp '=' ^+ '1'
        else
          ajoute_aux ap bp '-' ^+ '0'
      else if an = '0' && bm = '0' then
        if r = '+' then
          ajoute_aux ap bp '=' ^+ '1'
        else if r = '-' then
          ajoute_aux ap bp '+' ^+ '1'
        else
          ajoute_aux ap bp '=' ^+ '0'
      else (* an = 1 et bm = 0 ou l'inverse *)
        if r = '+' then
          ajoute_aux ap bp '-' ^+ '0'
        else if r = '-' then
          ajoute_aux ap bp '=' ^+ '0'
        else
          ajoute_aux ap bp '=' ^+ '1'
  in ajoute_aux a b '='
```

Ou en réutilisant la fonction de conversion déjà codée (attention, cette version est sujette au dépassement de capacité des entiers, ce qui n'est pas le cas pour la précédente) :

```
let from_digit c = int_of_char c - int_of_char '0'

let convertis_inv str b =
  let n = String.length str in
  assert (n > 0);
  let rec aux acc i =
    if i = n then
      acc
    else
      aux (acc * b + from_digit str.[i]) (i+1)
  in aux 0 0

let ajoute a b =
  let ap = convertis_inv a (-2) in
  let bp = convertis_inv b (-2) in
  convertis (ap + bp) (-2)
```

Numération bijective (11 pts)

Un système de numération bijectif est un système de numération dans lequel chaque nombre (entier positif) possède un unique représentant. Cette propriété n'est pas vérifiée pour la représentation décimale usuelle car le nombre 5 peut aussi être représenté 05 ou 005, etc. On va donc représenter les nombres sans utiliser de chiffre 0. La numération bijective en base k, ou notation k-adique, est une représentation en base k, mais qui utilise comme chiffres les nombres entiers de 1 à k. Pour $k = 10$ par exemple, on utilise un chiffre A dont la valeur est 10. Par exemple, le nombre 20 sera représenté $\underline{1A}_{10}$ car $20 = 1 \times 10 + A \times 1$. La représentation du nombre 179 reste inchangée car elle ne contient pas le chiffre 0.

☐ 34 – Compter jusqu'à 10 en notation 2-adique, en utilisant les chiffres 1 et 2.

Réponse 34. 1; 2; 11; 12; 21; 22; 111; 112; 121; 122

☐ 35 – Écrire le nombre 2010 en notation 10-adique, en utilisant les chiffres 1, 2, 3, 4, 5, 6, 7, 8, 9 et A.

Réponse 35. $2010 \to 1A10 \to 1A0A \to 19AA$

☐ 36 – Proposer un algorithme pour déterminer la notation k-adique d'un nombre.

Réponse 36. On procède par divisions euclidiennes successives, mais en modifiant la division pour garder un reste entre 1 et k inclus.

Le système bijectif en base 26, aussi appelé base 26 sans zéro, utilise comme chiffres les 26 lettres de l'alphabet en majuscule. C'est le système de numérotation qui est utilisé pour les colonnes des tableurs par exemple.

☐ 37 – Combien de colonnes se trouvent à gauche de la colonne de la colonne "ABC".

Réponse 37.

$$\begin{aligned} 1 \times 26^2 + 2 \times 26 + 3 &= 676 + 52 + 3 \\ &= 731 \end{aligned}$$

La colonne "ABC" est la 731-ième colinne, il y en a 730 à sa gauche.

☐ 38 – Écrire une fonction récursive `num_col : string -> int` qui prend en entrée une chaîne de caractères ne contenant que des lettres majuscules et qui renvoie le numéro de la colonne correspondante.
Si la chaîne de caractère contient un caractère autre qu'une lettre majuscule, on pourra lever une exception appropriée avec l'expression

```
raise (Invalid_argument "caractère illégal")
```

Réponse 38.

```
let letter c =
  if c < 'A' || c > 'Z' then
    raise (Invalid_argument "caractère illégal")
  else
    int_of_char c - int_of_char 'A' + 1

let num_col str =
  let n = String.length str in
  if n = 0 then 0 else
    let rec num_col_aux acc i =
      if i = n then
        acc
      else
        num_col_aux (acc * 26 + letter str.[i]) (i+1)
    in
    num_col_aux 0 0
```

Notation ternaire équilibrée (2 pts)

Le système ternaire équilibré consiste en un système de numération en base 3 dont les chiffres sont $+1$, 0, et -1, que nous noterons `+`, `0`, et `-`. Par exemple [`+--0`] correspond à $27 - 9 - 3 = 15$.

☐ 39 – Écrire le nombre 42 en ternaire équilibré.

Réponse 39.

$$\begin{aligned} 42 &= 14 \times 3 + 0 \\ 14 &= 5 \times 3 + (-1) \\ 5 &= 2 \times 3 + (-1) \\ 2 &= 1 \times 3 + (-1) \\ 1 &= 0 \times 3 + 1 \end{aligned}$$

Donc $42 =$ [`+---0`]. Vérification : $81 - 27 - 9 - 3 = 42$.

Partie VII. Anagrammes (OCaml – 5 pts)

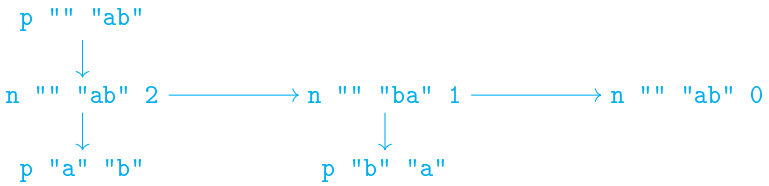
On donne les fonctions suivantes qui permet d'afficher toutes les anagrammes d'une chaîne de caractères :

```
let rec palindrome prefixe str =
  if String.length str <= 1 then
    print_endline (prefixe^str)
  else
    next_letter prefixe str (String.length str)

and next_letter prefixe str n =
  if n = 0 then
    ()
  else
    let premiere_lettre = String.make 1 str.[0] in
    let reste = String.sub str 1 (String.length str - 1) in
    palindrome (prefixe^premiere_lettre) reste;
    next_letter prefixe (reste^premiere_lettre) (n-1)
```

☐ 40 – Dessiner l'arbre des appels pour `palindrome "" "ab"`. On pourra noter p un appel de la fonction `palindrome` et n un appel de la fonction `next_letter`.

Réponse 40.



☐ 41 – Montrer que les fonctions `palindrome` et `next_letter` terminent.

Réponse 41. Chaque chemin dans l'arbre des appels contient des appels p espacés par des appels n. Lors des suites d'appel n, le 3-ème argument est un entier qui décroît strictement, donc ces suites sont nécessairement finies. Si on ignore les appels n le long d'un chemin, la taille du second argument des appels p est un entier strictement décroissant, donc ces suites sont nécessairement finies.
Ainsi, toutes les branches de l'arbre des appels récursifs sont finies et les fonctions terminent.

Fin de l'épreuve