TP 2 : Itération et récursion
(première partie)

Nous allons dans ce TP mettre en pratique ce que nous avons vu en cours sur les boucles et la récursion. Chaque exercice peut bien entendu être résolu de bien des façons. Essayez de réfléchir à plusieurs solutions, et n'oubliez pas, même si cela ne vous l'est pas systématiquement demandé, de démontrer la validité de vos programmes et d'en évaluer les complexités temporelle et spatiale.

Exercice 1 :

Écrire une fonction `somme_tableau : int vect -> int` qui lorsqu'on lui donne un tableau T d'entiers renvoie la somme de ses éléments.

Exercice 2 :

Écrire une fonction `recherche_tableau : int vect -> int -> bool` qui lorsqu'on lui donne un tableau T d'entiers et un entier n renvoie `true` si n se trouve dans T, `false` sinon,

- avec une boucle indexée
- avec une boucle conditionnelle
- (plus dur) à l'aide d'une fonction récursive.

Évaluez l'efficacité de chacune de ces fonctions.

Exercice 3 :

On représente les polynômes à coefficients entiers par des tableaux d'entiers.

1. Écrire une fonction `deg : 'a vect -> int` renvoyant le degré d'un polynôme (la longueur du vecteur) et une fonction `init_poly : int -> int vect` qui crée un polynôme de degré n donné initialisé à 0 (attention : celui-ci n'est pas un vrai polynôme de degré n !).
2. Écrire une fonction `add_p : 'int vect -> 'int vect -> 'int vect` qui renvoie la somme de deux polynômes. Écrire de même des fonctions `sub_p`, `mul_p`, `quo_p` et `res_p` renvoyant respectivement la différence, le produit, le quotient (euclidien) et le reste de la division euclidienne de deux polynômes.
3. Écrire une fonction `pgcd_p : 'int vect -> 'int vect -> 'int vect` calculant le pgcd de deux polynômes.
4. Évaluer les complexités temporelles et spatiales de toutes ces procédures.

On verra dans un prochain TP un moyen de manipuler efficacement des polynômes du genre $\mathrm{X}^{1000} - 1$, dont le stockage dans un tableau constitue un effroyable gaspillage d'espace.

Exercice 4 (Crible d'Eratosthène) :

1. Un entier naturel n étant donné, on veut tous les nombres premiers qui lui sont inférieurs ou égaux.

   On utilise pour cela la méthode du crible d'Eratosthène :

   - On écrit tous les nombres de 2 à n.
   - On cherche le premier nombre non rayé (au départ 2), et on raye tous ses multiples.
   - On recommence l'étape précédente jusqu'à ce qu'on ait exploré toute la table. Les nombres non barrés sont alors les nombres premiers inférieurs à n.

   Écrivez une fonction `crible : int -> unit` qui écrit à l'écran tous les nombres premiers inférieurs ou égaux à l'argument.

2. En constatant que mis à part 2, tous les nombres premiers sont impairs, et que si p est premier, le premier multiple de p non déjà rayé est p^2, améliorez la fonction précédente.

3. **Application graphique**

   Cet question relève plutôt du sujet d'étude. Comptez au moins une heure et demi pour la mise au point.

   La *spirale d'Ulam* est un tableau dans lequel les nombres entiers sont écrits en escargot, et dont les nombres premiers sont noircis (voir figure ci-dessous). Les alignements constatés correspondent à des valeurs de polynômes à coefficients entiers.

17	16	15	14	13
18	5	4	3	12
19	6	1	2	11
20	7	8	9	10
21	22	23	24	···

   Écrivez une fonction utilisant la bibliothèque graphique, et traçant une telle figure.

Exercice 5 (Tranche minimale d'un tableau d'entiers) :

Considérons un tableau A d'entiers relatifs, $A = (a_1, \ldots, a_n)$. On appelle *tranche* ou *coupe* de A une suite extraite notée $A[i..j]$ où $i \leqslant j$ et $A[i..j] = (a_i, \ldots, a_j)$. Nous noterons $\sigma(i,j)$ la somme des termes de la tranche $A[i..j]$:

$$\sigma(i,j) = \sum_{k=i}^{j} a_k$$

On s'intéresse à la recherche du minimum des $\sigma(i,j)$ pour $1 \leqslant i \leqslant j \leqslant n$.

1. On considère pour commencer l'algorithme naïf explorant toutes les tranches possibles :

   $S := a_1$
   pour j de 1 à n faire
   | pour i de 1 à j faire
   | | $S := min(S, \sigma(i,j))$
   | fin pour
   fin pour
   renvoyer S

   Le traduire en une fonction `tranche_naif : int vect -> int`.

   Étudiez la complexité de cet algorithme. Vous paraît-elle satisfaisante ?

2. En constatant que dans l'algorithme naïf, on calcule plusieurs (!) fois les mêmes sommes. Or ce sont ces sommes qui prédominent dans le calcul de complexité. Modifiez votre algorithme de sorte que dans la boucle interne le calcul de $\sigma(i,j)$ soit remplacé par une simple addition, et calculez la nouvelle complexité.

   Traduisez cet algorithme en une nouvelle fonction `tranche_moins_naif`.

3. On essaie d'améliorer notre analyse du problème en introduisant un raisonnement par récurrence. Considérons donc la suite $A_n = (a_1, \ldots, a_n)$ et supposons $n \geqslant 2$. Les tranches de A_n sont de deux types :

   - celles qui ne contiennent pas a_n : ce sont les tranches de A_{n-1} ;
   - celles qui contiennent a_n : il s'agit de la tranche $A_n[n..n]$ et des tranches de la forme $A_n[i..n]$ $(i < n)$ qui s'obtiennent en ajoutant a_n à la tranche $A_{n-1}[i..n-1]$ de A_{n-1}.

   Ceci nous invite à introduire la notion de *tranche finale* pour désigner les tranches dont le deuxième indice est l'indice du dernier terme de la suite considérée. Notons alors $m(A_n)$ le minimum des sommes des tranches de A_n (c'est ce qui nous intéresse) et $mf(A_n)$ le minimum des sommes des tranches finales. On a alors :

$$m(A_1) = mf(A_1) = a_1 \text{ et } \begin{cases} m(A_n) = \min(mf(A_n), m(A_{n-1})) \\ mf(A_n) = min(mf(A_{n-1}) + a_n, a_n) \end{cases}$$

   Implémentez cet algorithme en une fonction `tranche_recur`, et déterminez la nouvelle complexité.

4. Le dernier calcul ne doit pas nous faire perdre espoir : il suffit de l'envisager sous un angle itératif. Construisez un algorithme non récursif exploitant les constatations précédentes, et traduisez-le en une fonction `tranche_qui_decoiffe`. Vous devriez trouver une complexité en $O(n)$.

Sujet d'étude 1 (Manipulation de grands entiers) :

Ce sujet d'étude est relativement long. S'il vous intéresse, il vous faudra sans doute plusieurs heures de préparation avant d'obtenir les fonctions nécessaires à la réalisation de l'objectif final. Vous remarquerez sans doute que beaucoup des algorithmes construits dans l'exercice 3 sont presque immédiatement réutilisables. Ceci n'est bien entendu pas un hasard.

On représente donc les entiers par des tableaux (de taille fixée au départ) de *chiffres*, ces chiffres étant dans une certaine base b. Pour se fixer les idées, on prendra $b = 10000$, ce qui fait que chaque case du tableau correspondra à 4 chiffres de l'écriture décimale du nombre représenté.

1. Écrire les fonctions arithmétiques de base, ainsi que des fonctions de test ou d'initialisation qui peuvent vous sembler pertinentes. Indiquez les complexités de chacune de ces fonctions.

2. Testez vos fonctions en calculant 1000!, ou C_{533}^{227}. Essayez de trouver expérimentalement la complexité de vos fonctions, en les testant sur une collection de données.

3. **Une application : calcul d'une valeur approchée de π**

   Pour représenter un réel, on écrit le début de son développement décimal dans un entier long, la case de numéro 0 correspondant à la partie entière du réel.

   Archimède a calculé des valeurs approchées de π en approchant le périmètre d'un cercle de rayon 1/2 par le périmètre de polygones réguliers à 2^n cotés. Si p_n est ce périmètre, on a :

$$p_1 = 2, \quad p_{n+1} = \frac{p_n}{c_{n+1}} \text{ avec } c_1 = 0 \text{ et } c_{n+1} = \sqrt{\frac{1 + c_n}{2}}$$

$$0 < \pi - p_n \leqslant \frac{1}{6.4^{n-3}}$$

   En utilisant notre représentation des réels et la méthode d'Archimède, calculer quelques (un petit millier de ?) décimales de π.

Corrigé (partiel) du TP 2 : itération et récursion

Exercice 1

Exercice 2

Exercice 3

Exercice 4 :

1. Voici un programme qui imprime le résultat du crible. Vous trouverez en commentaires des instructions permettant de contrôler le bon fonctionnement du programme... ou de corriger un dysfonctionnement (beaucoup plus fréquent, d'après la célèbre loi de Murphy) (voir le programme 1).

Programme 1 Crible d'Eratosthène (avec contrôle du fonctionnement)

```
(* Crible d'Eratosthène *)
let eratosthene = function n ->
  let crible = make_vect (n+1) true
  and r = int_of_float(sqrt(float_of_int n))
  and compte = ref 0
  in
    begin
      for i = 2 to r do
        if crible.(i) then (* Si i n'a pas été rayé (il est premier) *)
          begin            (* on raye tous les multiples de i *)
            (* print_string("je raye les multiples de ");
               print_int(i);
               print_newline(); *)
            let j = ref (i * i) (* à partir de i^2 *)
            and pasfini = ref true
            in
              while !pasfini do
                crible.(!j) <- false;
                (* print_int(i);
                   print_string("  "); *)
                compte := !compte + 1;
                j := !j + i;
                pasfini := (n >= !j); (* et ce jusqu'à ce que j *)
              done;                   (* soit supérieur à n *)
              (* print_newline(); *)
          end
      done;
      for i = 2 to n do
        if crible.(i) then
          begin
            print_int(i);
            print_string("  ");
          end
      done;
      print_newline();
      (* print_string(" j'ai rayé ");
         print_int(!compte);
         print_string(" nombres.");
         print_newline(); *)
    end;;
```

2. Le programme précédent tient compte des remarques de cette question.

3. Le programme 1 utilise les capacités graphiques de l'interface graphique. A priori, si j'ai correctement lu la documentation, il doit tourner sans problèmes sous Windows.

`Programme 2 spirale d'Ulam : la fonction crible`

```
#open "graphics";;
open_graph " 501x501+50+50";;

let eratosthene = function n ->
  let crible = make_vect (n+1) true
  and r = int_of_float(sqrt(float_of_int n))
  and compte = ref 0
  in
    begin
      for i = 2 to r do
        if crible.(i) then (* Si i n'a pas été rayé (il est premier) *)
          begin           (* on raye tous les multiples de i *)
            (* print_string("je raye les multiples de ");
               print_int(i);
               print_newline(); *)
            let j = ref (i * i) (* à partir de i^2 *)
            and pasfini = ref true
            in
              while !pasfini do
                crible.(!j) <- false;
                (* print_int(i);
                   print_string("  "); *)
                compte := !compte + 1;
                j := !j + i;
                pasfini := (n >= !j); (* et ce jusqu'à ce que j *)
              done;                   (* soit supérieur à n *)
              (* print_newline(); *)
          end
      done;
      crible;
    end;;
```

Il utilise une version modifiée du crible d'Eratosthène (le programme 1). L'effet est joli, mais on pourrait l'améliorer en dessinant des points plus gros, ou bien en changeant la forme de la spirale. Les alignements diagonaux correspondent aux valeurs prises par des polynômes à coefficients entiers (lesquels ?).

Sujet d'étude 5 :

Nous corrigerons ce sujet d'étude en TD de façon détaillée.

Exercice 1

Programme 3 spirale d'Ulam : la fonction de tracé

```
let ulam () =
  let u = ref 250
  and v = ref 250
  and n = ref 1
  and crible = eratosthene 251001
  in
    for i = 1 to 250 do
      u := !u + 1;
      n := !n + 1;
      if crible.(!n) then plot !u !v;
      for j = 1 to (2 * i - 1) do
        n := !n + 1;
        v := !v + 1;
        if crible.(!n) then plot !u !v;
      done;
      for j = 1 to (2 * i) do
        n := !n + 1;
        u := !u - 1;
        if crible.(!n) then plot !u !v;
      done;
      for j = 1 to (2 * i) do
        n := !n + 1;
        v := !v - 1;
        if crible.(!n) then plot !u !v;
      done;
      for j = 1 to (2 * i) do
        n := !n + 1;
        u := !u + 1;
        if crible.(!n) then plot !u !v;
      done;
    done;;
```