Programmation C
TP n° 2 : tableaux, `enum`, `struct`

Les exercices indiqués avec une $*$ sont à rendre.

Exercice 1 : Fonctions avec des tableaux en argument

- Écrire une fonction `void somme(int T[], int S[], size_t n)` qui prend en arguments deux tableaux de taille `n` et qui modifie `S` de manière à ce que `S[i]` soit égale à la somme des éléments `T[0]`$+\ldots+$`T[i]` pour tout `i` inférieur à `n`.
- Écrire une fonction `void permute(int T[],size_t n)` qui prend en argument un tableau de taille `n` et qui modifie `T` en plaçant la valeur `T[0]` en position `1`, `T[1]` en position `2`,...,`T[n-2]` en position `n-1`, et `T[n-1]` en position `0`.
  Par exemple, si `T={2,5,1,6,8,4,9,10,12,7}`, alors après `void permute(T,10)` le tableau `T` est de la forme `T={7,2,5,1,6,8,4,9,10,12}`

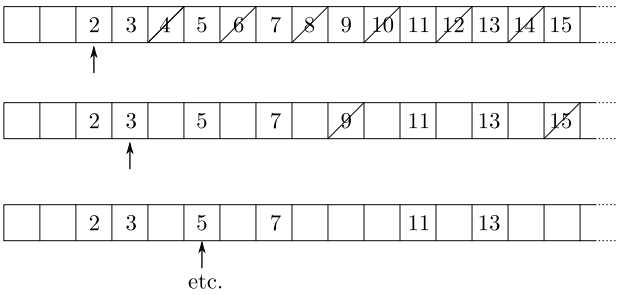
On testera les deux fonctions sur des tableaux de différentes tailles...

Exercice 2 : crible d'Ératosthène ($*$)

Le *crible d'Ératosthène* est une méthode permettant de calculer tous les nombres premiers inférieurs à un entier *SUP* donné. Son principe est le suivant :

1. On écrit la liste de tous les entiers supérieurs ou égaux à 2 et inférieurs à *SUP*.
2. On effectue un parcours de cette liste. À chaque entier i rencontré, on supprime de la liste tous les entiers strictement plus grands que i et multiples de i encore présents.

En fin de traitement, les nombres encore présents dans la liste sont tous les nombres premiers inférieurs à *SUP*. Voici par exemple l'état de la liste après la rencontre de 2, puis 3 – à la dernière étape, la rencontre de 5 sera suivie de l'effacement de tous les multiples de 5 encore présents (*e.g.* 25), etc.



Implémentation de l'algorithme en C. L'entier *SUP* peut être représenté par une constante `SUP` supérieure ou égale à 2, définie par un `#define` en début de programme. La liste peut être représentée par un simple tableau de booléens (`bool`, ne pas oublier de mettre `#include <stdbool.h>` en début de fichier) à `SUP` éléments, initialisé par des `true` (tous les nombres sont des "candidats" potentiels). À chaque étape du traitement, on peut faire la convention suivante :

- A partir de la position 2, si la case de position `i` contient un `true`, alors l'entier `i` est encore présent dans la liste représentée par le tableau. Si elle contient la valeur `false`, alors l'entier `i` a été supprimé de la liste (c'est un multiple d'un nombre inférieur).

Autrement dit, la notion de "suppression de l'entier i dans la liste" est représentée par l'écriture d'un `false` à la position `i` dans le tableau.

En fin de traitement, à partir de la position `2`, les valeurs de positions contenant encore un `true` seront celles de tous les nombres premiers inférieurs à `SUP`.

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 ...
```

On écrira trois fonctions :

- `void initialisation(bool T[],size_t n)` qui initialise les `n` cases du tableau `T` avec `true`.
- `void remplissage(bool T[],size_t n)` qui remplit le tableau en suivant les indications précédentes.
- `void affichNbPremier(bool T[],size_t n)` qui affiche les nombres premiers selon le tableau `T` rempli par la fonction précédente.

Dans le `main`, on déclarera un tableau de taille `SUP` et on appellera les deux fonctions précédentes.

Remarque. Un entier j est multiple de i si et seulement si j modulo i est égal à 0. (l'opérateur de modulo s'écrit `%` en C), mais on a aussi : un entier $j > i$ est multiple de i si et seulement s'il est de la forme $2 \times i$, $2 \times i + i$, $2 \times i + i + i$ ....

Exercice 3 : Récursivité et déclaration de fonction

On définit les deux fonctions `f` et `g` sur les entiers de la manière suivante :

$$\mathtt{f}(n) = \begin{cases} 2 & \text{si } n = 1 \\ 2 \cdot \mathtt{g}(n-1) & \text{sinon} \end{cases} \qquad \mathtt{g}(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3 \cdot \mathtt{f}(\frac{n}{2}) & \text{sinon} \end{cases}$$

Faire un programme C pour ces deux fonctions. Donner la valeur de $\mathtt{f}(20)$.

Exercice 4 : Fractions avec `struct` (∗)

Dans cet exercice, on se propose de définir un type avec `struct` afin de représenter les rationnels sous forme de fractions. Pensez à tester votre code à chaque question !

1. À l'aide de `struct`, définissez un type de structure `fraction` avec deux champs entiers `num` et `den` de type `long int` qui représentent respectivement le numérateur et le dénominateur de la fraction. Utilisez le mot clé `typedef` afin de créer l'alias `fraction` pour le type `struct fraction` et rendre ainsi votre code plus lisible.
2. Écrivez une fonction d'en-tête `fraction build(long int n, long int d)` qui prend en arguments deux entiers `n` et `d` et qui retourne la fraction $\frac{n}{d}$.

La fonction définie à la question précédente à un défaut évident : elle permet de construire des fractions avec 0 comme dénominateur ! Afin de pallier ce problème, on se propose d'utiliser la fonction `assert` de la bibliothèque standard. Une commande de la forme `assert(b)` évalue la condition logique `b`. Si la condition est fausse, le programme interrompt son exécution en affichant un message d'erreur. Pour vous servir de cette fonction, il faut ajouter au début de votre programme la ligne suivante :

```
1 #include <assert.h>
```

3. Modifiez le code de la fonction `build` afin de provoquer une erreur si l'on essaye de construire une fraction dont le dénominateur est `0`. Ensuite, dans le `main`, créez un tableau `ex_fractions` de fractions qui contient les fractions $\frac{1}{1}$, $\frac{1}{2}$, $\frac{2}{4}$, $\frac{-9}{3}$, $\frac{8}{-20}$, $\frac{-5}{-1}$, $\frac{1}{-3}$.

4. Écrivez une fonction d'en-tête `bool eq_fraction(fraction f, fraction g)` qui renvoie `true` si les deux fractions sont égales et `false` sinon. On rappelle que deux fractions $\frac{a}{b}$, $\frac{c}{d}$ sont égales si et seulement si $a * d = c * b$.

5. Écrivez une fonction `bool is_int(fraction f)` qui renvoie `true` si la fraction `f` est un entier (c'est-à-dire peut être mis sous la forme $\frac{n}{1}$ où n est un entier) et `false` sinon.

6. Écrivez les fonctions suivantes qui calculent la somme, la soustraction et la multiplication de fractions.

```
1       fraction sum(fraction f, fraction g) // somme
2       fraction sub(fraction f, fraction g) // soustraction
3       fraction mul(fraction f, fraction g) // multiplication
```

7. Écrivez une fonction d'en-tête `fraction reduce(fraction f)` qui renvoie la fraction `f` sous forme irréductible. Pour cela, on pourra d'abord coder la fonction `long pgcd(long a, long b)` qui calcule le pgcd des deux entiers `a` et `b`. On rappelle que l'algorithme d'Euclide pour calculer le pgcd de deux entiers *positifs* a et b est (en pseudo-code) :

```
x <- a
y <- b
while (y !=0){
  r <- reste de la division euclidienne de x par y
  x <- y
  y <- r
}
return x
```

Vous devez également faire en sorte que lorsque la fraction renvoyée par `reduce` est négative, le signe apparaisse au numérateur et non pas au dénominateur. Testez vos fonctions sur les fractions de `ex_fractions`.