TP n°7 - Approfondissement OCaml.

Exercice 1 (Algorithmes de recherche).
On codera les fonctions de cet exercice en OCaml dans un fichier `algo_tableaux.ml`.
On aura pris soin de recopier la fonction `tri_a_bulles` du DM1 dans ce nouveau fichier.
Un algorithme de recherche est un algorithme dont l'objectif est :

— de déterminer si une donnée appartient ou non à un ensemble de données

— de donner au moins une localisation de cette donnée (la première trouvée)

1. Coder l'algorithme naïf `seq_search` de recherche séquentielle d'une valeur dans un tableau quelconque dans une fonction. Testez le sur au moins 3 tableaux différents.

2. Quel est le prototype inféré pour votre fonction ? Quelle possibilité du langage OCaml est ici utilisée ?

3. Nous allons maintenant coder l'algorithme de recherche dichotomique (*binary search* en anglais).

Pour cet algorithme, on suppose que le tableau donné en entrée de l'algorithme est déjà trié dans l'ordre croissant.

L'algorithme est itératif : à chaque itération, on regarde la valeur médiane du tableau :

— Si la valeur recherchée est égale à la valeur médiane du tableau, c'est gagné.

— Si la valeur recherchée est strictement plus grande que cette valeur médiane, alors il faut continuer à chercher dans la partie droite du tableau.

— Sinon, il faut rechercher dans la partie gauche du tableau.

Coder cet algorithme dans une fonction `bin_search` en utilisant le paradigme de programmation impératif.

4. Comparer les complexités dans le pire des cas des fonctions `seq_search` et `bin_search`. Quelle est le meilleur algorithme pour les grands tableaux ?

Exercice 2 (Fonctions récursives).
Toutes les fonctions ci-dessous seront codées en OCaml dans un fichier `recursivite.ml`.

1. Écrire une fonction `suite_u` capable de calculer n'importe quel terme u_n de la suite $(u_n)_{\mathbb{N}}$ définie par :

$$\begin{aligned} u_0 &= 1 \\ u_n &= \sqrt{u_{n-1}+2} \quad \forall n > 0 \end{aligned}$$

Indication : penser à utiliser la syntaxe de filtrage et les exceptions pour commencer à produire des codes OCaml de qualité gérant tous les cas.

Tester la fonction, valider les types inférés. Vérifier les valeurs obtenues en comparant avec celles obtenues avec la calculatrice. Cette suite semble-t-elle converger ?

2. Écrire une fonction récursive `somme_recursive` qui calcule la somme des n premiers entiers de manière récursive

3. Écrire une fonction itérative naïve `power` qui calcule x^n à partir de la donnée de x et de n.

4. Écrire une fonction récursive `power_bin` qui calcule x^n en utilisant les formules suivantes :

$$\begin{aligned} x^{2n} &= (x^n)^2 \\ x^{2n+1} &= (x^n)^2 \times x \end{aligned}$$

5. Comparer `power_bin` et `power` en terme de complexité temporelle (nombre d'opérations élémentaires) et spatiale (encombrement mémoire) en fonction de la puissance n donnée en entrée. Réponse à rédiger sur papier pour s'entraîner pour le DS.

6. (A faire à la maison pour s'entraîner) Créer un fichier `recursivite.c` codant et testant toutes ces fonctions en langage C. Vous mettrez tous vos tests pour les différentes fonctions "en dur" dans le main, sans passer par la ligne de commande. La fonction racine carrée se trouve dans la bibliothèque `math.h` et se nomme `sqrt`. Il faut également rajouter l'option `-lm` sur la ligne de commande lors de la compilation pour utiliser cette fonction.

Exercice 3 (Nombres rationnels).
Le but de cet exercice est créer une petite bibliothèque permettant de manipuler des fractions. Une fraction $\frac{a}{b}$ est représentée par deux entiers, le numérateur a et le dénominateur b. On souhaite de plus avoir les propriétés suivantes :

— la fraction doit toujours être donnée sous sa forme irréductible, c'est-à-dire telle que le PGCD de a et de b vaut 1 (pas de diviseurs communs)

— b est toujours positif, autrement dit, le signe de la fraction est celui du numérateur

1. Écrire une fonction OCaml `pgcd: int → int → int` qui calcule le PGCD de deux entiers en utilisant le fait que le PGCD de a et de b est égal au PGCD du reste de la division euclidienne de a par b et de b

$$PGCD(a, b) = PGCD(amodb, b)$$

2. Définir un type enregistrement `frac` OCaml permettant de définir des fractions.

3. Écrire une fonction OCaml `simp` qui simplifie une fraction et s'assure que le dénominateur est toujours positif. **Toutes les fonctions qui suivent doivent renvoyer des fractions simplifiées respectant les critères énoncés.**

4. Écrire une fonction OCaml `creer_frac` qui crée une fraction à partir de la donnée de a et de b.

5. Écrire une fonction OCaml `add_frac` qui renvoie la somme de deux fractions.

6. Écrire une fonction OCaml `mult_frac` qui renvoie le produit de deux fractions.

7. Écrire une fonction OCaml `div_frac` qui renvoie le quotient de deux fractions.

8. Écrire une fonction OCaml `inv_frac` qui renvoie l'inverse d'une fraction

9. Écrire une fonction OCaml `neg_frac` qui renvoie l'opposé d'une fraction

10. Écrire une fonction OCaml `float_of_frac` qui convertit une fraction en nombre flottant

11. Écrire une fonction OCaml `string_of_frac` qui convertit une fraction en chaîne de caractère

Exercice 4 (Ordre lexicographique, fonction d'ordre supérieur).
Dans cette exercice, nous allons donner une relation d'ordre sur l'ensemble des chaînes de caractères. Autrement dit, nous allons donner une définition possible pour exprimer le fait qu'une chaîne de caractères est strictement inférieure à une autre chaîne de caractères.

L'**ordre lexicographique** est la relation d'ordre utilisée pour ranger les mots dans un dictionnaire occidental. Les mots sont rangés par ordre alphabétique : d'abord ceux commençant par la lettre a, puis ceux commençant par b...etc Pour ranger deux mots qui commencent par la même lettre, on s'appuie sur la deuxième lettre. Puis pour ranger deux mots commençant par les deux mêmes premières lettres, on s'appuie sur la troisième lettre...etc

On note $\prec$ la relation d'ordre entre deux chaînes de caractères, celle à gauche du symbole étant rangée avant celle de droite dans l'ordre lexicographique.

Par exemple, les relations d'ordre suivantes sont vraies :

$$\texttt{abeille} \prec \texttt{atelier}$$
$$\texttt{tabulation} \prec \texttt{zebulon}$$
$$\texttt{albatros} \prec \texttt{albigeois}$$
$$\texttt{tab} \prec \texttt{tabulation}$$

L'**ordre lexicographique est une relation d'ordre totale** : pour deux chaînes de caractères, il est toujours possible de les comparer, c'est à dire de dire si elles sont égales ou si l'une est inférieure à l'autre au sens de la relation d'ordre lexicographique.
Dans la suite, on codera les fonctions dans un fichier `ordre_lexicographique.ml`.

1. Classez les chaînes de caractères suivantes dans l'ordre lexicographique : mixtape, masochisme, deschamps, miserable, deshydratation, retourner, afficher, different
2. Écrire une fonction **itérative** `ordre_lexico_strict_inf` comparant deux chaînes de caractères. Elle doit répondre vrai si la première chaîne de caractères est inférieure strictement à la seconde pour la relation d'ordre lexicographique. Elle doit répondre faux dans le cas contraire, ou si les deux chaînes sont identiques.
3. Adaptez la fonction de tri à bulles en créant une nouvelle fonction `tri_a_bulles_generique` pour qu'elle puisse ranger des objets d'un type quelconque par ordre croissant pourvu qu'on lui fournisse une relation d'ordre sur ce type d'objets. *Indication : fonction d'ordre supérieur.*
4. Testez votre tri générique sur des tableaux de chaînes de caractères

L'ordre lexicographique est celui utilisé par défaut par OCaml pour comparer des chaînes de caractères. Le test de strict infériorité entre deux chaînes de caractères est tout simplement noté avec le symbole `<`*. Par exemple, l'évaluation de l'expression*

```
''toto'' < ''tota'';;
```

par l'interpréteur donnera la valeur `false`*. En d'autres termes,* `<` *est un opérateur polymorphe en OCaml : il permet de comparer des objets de différents types : entiers, flottants, caractères, chaînes de caractères... Par contre, en C, il est obligatoire d'implémenter un ordre, par exemple l'ordre lexicographique, si l'on souhaite ordonner des chaînes de caractères.*

Exercice 5 (DM : Base de donnée de véhicules en circulation).
Toutes les fonctions de cette exercice seront codées dans un fichier `vehicule.ml`.

1. Créer un nouveau type `vehicule_en_circulation` permettant de stocker les informations relatives à un véhicule en circulation en France :

- nom du propriétaire,
- description du véhicule : type, marque et puissance.

  Soit le véhicule est une voiture, de marque parmi Renault, Peugeot, Citroën ou une autre marque étrangère, en précisant à chaque fois le modèle et le nombre de chevaux du moteur.
  Soit le véhicule est une moto : dans ce cas, on ne précise pas la marque mais on donne simplement la cylindrée (50 cm3, 125 cm3, 250 cm3 ou une cylindrée supérieure à 250 cm3).
  Vous créerez autant de types que nécessaire.
- plaque d'immatriculation (deux lettres + trois chiffres + deux lettres)

2. Créer un base de donnée de tests en créant un tableau contenant les données de 5 véhicules en circulation. Vous inventerez les informations

3. Créer une fonction permettant d'ordonner les objets du type `vehicule_en_circulation` selon l'ordre lexicographique appliqué au nom du propriétaire. On généralisera la fonction `tri_a_bulles` implémentée en DM pour lui permettre de connaitre l'ordre sur les objets de type `vehicule_en_circulation`. Peu de nouveau code est nécessaire.