Option informatique : TP 5
(Diviser pour Régner)

Des fonctions d'aide sont fournies à l'adresse suivante pour certaines questions : `https://who.rocq.inria.fr/Simon.Cruanes/enseignement/tp5_helper.ml`.

1 Tri Fusion

Nous allons commencer par implémenter le tri fusion sur les listes afin de bien assimiler la technique.
(a) Écrire une fonction `merge : 'a list -> 'a list -> 'a list` qui, à des entrées triées, associe une sortie triée.
(b) Écrire une fonction `split : 'a list -> ('a list * 'a list)` qui découpe une liste en deux sous-listes de longueur égales (à un élément près si la longueur est impaire).
(c) Écrire une fonction `merge_sort: 'a list -> 'a list` qui utilise le principe du « diviser pour régner » et les deux fonctions précédentes.
(d) On peut écrire une fonction qui génère des listes aléatoires d'entiers dans $[\![0, \text{maxval}]\!]$ de longueur n, et une fonction pour mesurer le temps que prend une fonction f à un argument de la manière suivante :

```
let rec randlist maxval n = match n with
    | 0 -> [] |
    _ -> random__int maxval :: randlist (n-1);;

let mesurer_temps f x =
    let n = 20 in  (* nombre d'iterations *)
    let start = sys__time () in
    for i = 1 to n do
        let _ = f x in
        ()
    done;
    let stop = sys__time () in
    (stop -. start) /. (float_of_int n);;
```

Avec cela, mesurer le temps pris par la fonction `merge_sort` sur des entrées de plus en plus grande, et comparer avec `insert_sort` dont la définition suit.

```
let rec insert x l = match l with
    | [] -> [x]
    | y::l' -> if x <= y then x :: l else y :: insert x l';;

let rec insert_sort l = match l with
    | [] -> []
    | x :: l' -> insert x (insert_sort l');;
```

2 Exponentiation Rapide

L'opération $x \mapsto x^n$ peut être très coûteuse si on l'implémente naïvement. Nous allons étudier une version « diviser pour régner » plus efficace, appelée « exponentiation rapide ».

2.1 Version Simple

(a) Commençons par écrire une version naïve de l'exponentiation, `puissance_naive: int -> int -> int` telle que `puissance_naive x n` calcule x^n en $O(n)$ multiplications.
(b) En exploitant la propriété

$$x^n = \begin{cases} (x^2)^{\frac{n}{2}} & \text{si } n \equiv 0 \ [2] \\ x \cdot (x^2)^{\frac{n-1}{2}} & \text{sinon} \end{cases}$$

écrire une version efficace de l'exponentiation. Cette version est appelée *exponentiation rapide*.
(c) Montrer que cette seconde version effectue $O(ln(n))$ produits (et appels récursifs) pour calculer x^n.

2.2 Version Générique

Dans la suite nous allons écrire des fonction *génériques*, dans le sens qu'elles sont abstraites et fonctionnent sur n'importe quel type et produit, du moment que produit forme un *monoïde*, c'est-à-dire qu'il est associatif et possède un élément neutre (le cas $n = 0$). De telles structures abondent, que ce soit les entiers, les chaînes, les listes (avec `@`), les polynômes, les matrices, etc. Nous allons définir un *type enregistrement* pour représenter la structure de monoïde.

```
type 'a monoide = {
    neutre : 'a;
    produit : ('a -> 'a -> 'a);
};;

let monoide_int = {
    neutre = 1;
    produit = (fun x y -> x*y);
};;

let monoide_string = {
    neutre = "";
    produit = (fun x y -> x^y);  (* concatenation *)
};;

(* acces aux elements du type enregistrement *)
let multiplier monoide x y = monoide.produit x y;;
```

```
multiplier monoide_int 10 20 = 200;;
multiplier monoide_string "hello " "world" = "hello world";;
```

Ici nous avons défini un type `'a monoide`, qui décrit un monoïde dont les éléments sont de type `'a`, et deux valeurs qui décrivent respectivement le produit sur les entiers, et la concaténation de chaînes (qui forment tout deux des monoïdes).
(a) Implémenter une fonction `puissance_naive2 : 'a monoide -> 'a -> int -> 'a` linéaire en temps, qui calcule la puissance $n^{\text{ème}}$ de son argument de type `'a`.
(b) Créer une chaîne contenant 1000 caractères `'a'` à la suite.
(c) Écrire une fonction d'exponentiation rapide générique (du même type que `puissance_naive2`).

2.3 Suite de Fibonacci : le Retour

Si on exclut la formule analytique permettant de calculer les termes de la suite de Fibonacci directement [1], la meilleure technique que nous ayons vu jusqu'ici était de complexité linéaire. Nous allons maintenant voir une version logarithmique.
(a) Montrer que $\forall n \in \mathbb{N}^*, \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^n = \begin{pmatrix}F_{n+1} & F_n\\ F_n & F_{n-1}\end{pmatrix}$ avec $F_0 = 0, F_1 = 1, F_{n+2} = F_{n+1} + F_n$ la suite de Fibonacci.
(b) On va représenter les matrices par un type `type matrix == int vect vect` (lignes, colonnes). Implémenter une fonction de type `matrix -> matrix -> matrix` qui, en supposant que le nombre de colonnes de son deuxième argument correspond au nombre de lignes de son premier argument, renvoie une nouvelle matrice qui est le produit des arguments. On notera que cette fonction est associative mais pas commutative. Quel est l'élément neutre associé au produit de matrices ?
On pourra utiliser la fonction `init_vect : int -> (int -> 'a) -> 'a vect` pour construire des matrices.
(c) En utilisant la fonction d'exponentiation rapide générique précédente, écrire une fonction qui calcule le $n^{\text{ème}}$ terme de la suite de Fibonacci en temps $O(ln(n))$.

3 Multiplication de Karatsuba

L'algorithme de Karatsuba permet de multiplier des nombres de taille n (nombre de chiffres) plus efficacement que l'algorithme « classique » de complexité $O(n^2)$. C'est utile en pratique pour implémenter des *entiers à précision arbitraire* (comme ceux de python). L'idée est la suivante : on choisit un naturel B^m et on décompose les entiers x et y à multiplier en $x = x_1 \times B^m + x_2$ et $y = y_1 \times B^m + y_2$ avec x_2 et $y_2 < B^m$. B est la base (typiquement 2 ou 10) et m l'exposant. Puis, on décompose $x \times y = (x_1 \times y_1) \times B^{2m} + (x_1 \times y_2 + y_1 \times x_2) \times B^m + x_2 \times y_2$. Cela fait donc quatre sous-produits à calculer (sans compter les produits par B^m ou B^{2m}). Soit $z_1 = x_1 \times y_1$ et $z_2 = x_2 \times y_2$. On peut se ramener à calculer trois

1. même en ce cas, le calcul de la puissance $n^{\text{ème}}$ n'est pas gratuite.

produits ainsi : $x_1 \times y_2 + y_1 \times x_2 = (x_1 + x_2) \times (y_1 + y_2) - z_1 - z_2$ avec z_1 et z_2 déjà calculés.

(a) Implémenter une fonction qui calcule, pour B et m fixés (par exemple $B = 2$ et $m = 10$, ce qui fait une base 1024), le produit de deux entiers naturels. On utilisera le produit classique `*` uniquement quand les entiers sont plus petits que B^m (ou pour calculer B^{2m}).

À la fin du TP n° 1, nous avons étudié les opérations sur des polynômes à coefficients entiers sur une variable, ceux-ci étant représentés par des tableaux non-vides de longueurs variables. Ainsi, le polynôme $a_n X^n + a_{n-1} X^{n-1} + \ldots + a_0$ (de degré n) est représenté par le tableau `[|`a_0`;`a_1`;...;`a_n`|]` (de longueur $n+1$). Nous allons calculer des puissances de polynômes par la méthode d'exponentiation rapide.

(b) Implémenter des fonctions de somme et différence de deux polynômes, de type `int vect -> int vect -> int vect`. Le degré d'un polynôme `p` est donné par `vect_length p - 1`.

note : on peut définir un opérateur infixe de la manière suivante (pour définir une notation pratique pour la somme :

```
let prefix ++ p q = somme_poly p q ;;

[| 1; 2 |] ++ [| 0; 4; 6 |] = [| 1; 6; 6; |];;
```

Il est fortement recommandé d'utiliser ce type de notations par la suite, en prenant garde à ne pas masquer les opérateurs normaux du langage. On trouvera divers opérateurs et des tests à l'adresse suivante.

(c) Implémenter le produit de deux polynômes de manière « classique » par la formule $(P \times Q)(X) = \sum_{i=0}^{m+n} \sum_{j=0}^{\min(i,m)} P_i \cdot Q_{i-j} \cdot X^i$ si $P(X) = \sum_{i=0}^{m} P_i \cdot X^i$ et $Q(X) = \sum_{i=0}^{n} Q_i \cdot X^i$.

(d) Implémenter le produit de deux polynômes en utilisant la méthode de Karatsuba et le précédent produit pour les cas terminaux (càd le produit de « petits » polynômes). On choisira une base de la forme $B(X) = X^m$ (avec typiquement m valant 5, ou 10), pour laquelle le quotient et le reste sont triviaux à calculer (sous-tableaux). On pourra utiliser les fonctions intermédiaires `shift_poly`, `quotient_poly` et `reste_poly` définies dans le fichier d'aide, respectivement pour multiplier un polynôme par X^m, obtenir son quotient et son reste par la division par X^m, et telles que pour tout polynôme `p` et entier `n > 0`, `p = somme_poly (shift_poly (quotient_poly p n) n) (reste_poly p n)` (c'est-à-dire $P = Q \times X^m + R$ quand Q, R sont le quotient et le reste de la division de P par X^m, avec $\deg(R) < m$).

note : il n'est pas forcément facile de rendre cette fonction efficace en pratique, à cause des nombreuses sommes et copies de polynômes.

(e) Implémenter l'exponentiation de polynômes et comparer ses performances à une version naïve. On remarquera que l'élément neutre du produit de polynômes est le polynôme constant de valeur 1. En appliquant cette fonction au polynôme $X + 1$ et à diverses puissances, que peut-on remarquer sur les coefficients ?