Exercice 1 :

1. Écrire (sans utiliser l'opérateur de comparaison sur les listes) une fonction récursive `comparaison : 'a list -> 'a list -> bool` qui prend en argument deux listes et indique si la première est strictement plus grande que la seconde dans l'ordre lexicographique (avec les préfixes considérés plus petits).
   Exemples :
   - `comparaison [1; 2; 3] [2; 5] = false` car $1 < 2$.
   - `comparaison [1; 2; 3] [1; 2; 3] = false` car les listes sont égales.
   - `comparaison [1; 2; 3] [1; 2] = true` car un préfixe strict est plus petit.
   - `comparaison [1; 3; 3] [1; 2; 5] = true` car $3 > 2$.

   À l'avenir, plutôt que d'implémenter cette fonction, vous pouvez simplement utiliser les opérateurs de comparaison sur les listes, qui correspondent alors au même ordre lexicographique.
2. (a) Écrire une fonction récursive `retire : 'a list -> 'a -> 'a list` telle que `retire l x` soit une liste qui contient les mêmes éléments que `l` dans le même ordre, sauf tous ceux de valeur `x`.
   Exemple : `retire [3; 0; -1; 1; 61; 2; 1; 1; 8; 14] 1 = [3; 0; -1; 61; 2; 8; 14]`.
   (b) Cette fonction a-t-elle un effet de bord ? Si on représente `l` et `retire l 1` par des maillons, les deux listes ont-elles des choses en commun ?
   (c) Donner une relation de récurrence sur le nombre d'opérations effectué par `retire` en fonction de la taille de `l`. En déduire l'ordre de grandeur de la complexité de `retire` en fonction de la taille de `l`.
3. En vous inspirant de la question précédente, écrire une fonction `indices_pairs : 'a list -> 'a` telle que `indices_pairs l` renvoie la liste $[x_0; x_2; \dots; x_{2\lfloor \frac{n-1}{2} \rfloor}]$, avec `l` $= [x_0; x_1; \dots; x_{n-1}]$.
   Exemple : `indices_pairs [3; 0; -1; 1; 61; 2; 1; 1; 8; 14] = [3; -1; 61; 1; 8]`
   Vérifier que votre fonction a bien une complexité linéaire en la taille de la liste.

Exercice 2 :

1. (a) Écrire une fonction récursive `concat : 'a list -> 'a list -> 'a list` qui prend en argument deux listes d'éléments de même type et renvoie une liste composée des éléments de la première suivi des éléments de la deuxième.
   Exemple : `concat [1; 42; 3] [3; 17; -2; 5] = [1; 42; 3; 3; 17; -2; 5]`.
   À l'avenir, plutôt que réécrire cette fonction vous pouvez utiliser l'opérateur `@`.
   (b) Comment sont représentées en mémoire les listes argument de `concat` et la liste résultat ?
   (c) Quelle est l'ordre de grandeur de la complexité de `concat` en fonction de n_1 et n_2 les tailles des listes en argument ?
2. On souhaite écrire une fonction pour calculer le renversé d'une liste (à l'avenir on pourra utiliser `List.rev` poue cela).
   (a) Pourquoi ne peut-on pas simplement écrire une fonction récursive `rev : 'a list -> 'a list` qui fait ce qu'on veut ?
   (b) Écrire une fonction récursive `rev_aux : 'a list -> 'a list -> 'a list` telle que `rev l1 l2` calcule le renversé de `l1` suivi de `l2`.
   Exemple : `rev_aux [1; 2; 42] [12; 18] = [42; 2; 1; 12; 18]`.
   (c) En utilisant la fonction ci-dessus, écrire `rev`.
3. Écrire une fonction récursive `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` telle que `fold_left f init l`, avec `l` $= [x_0; x_1; \dots; x_{n-1}]$ est `f (f ... (f init` x_0`) ...` x_{n-2}`)` x_{n-1}.
   Exemple : `fold_left min max_int [3; 17; -2; 5] = -2`.
   À l'avenir, plutôt que réécrire cette fonction vous pouvez utiliser la fonction `List.fold_left`.