Exercice 1.

Écrire une fonction « `test: int array -> bool` » de **complexité linéaire** qui prend en entrée un tableau `tab` de taille n et renvoie `true` lorsque tous les éléments de $[\![0, n-1]\!]$ apparaissent dans `tab`. Dans le cas contraire, votre fonction renverra `false`. Vérifiez en particulier que « `test [|-1; 2|]` » vaut `false`.

Exercice 2. Listes chaînées

Toutes les fonctions demandées dans cet exercice doivent être écrites directement sans définir de fonction intermédiaire.

1. Écrire une fonction `double` qui prend en entrée une liste `li` et renvoie la liste où chaque élément de `li` apparaît deux fois consécutivement. Par exemple, « `double [1; 2; 3; 4]` » vaut `[1; 1; 2; 2; 3; 3; 4; 4]`.
2. Écrire une fonction « `dernier: 'a list -> 'a` » qui renvoie le dernier élément de la liste en entrée.
3. Écrire une fonction « `avant_dernier: 'a list -> 'a` » qui renvoie l'avant-dernier élément de la liste en entrée.
4. Écrire une fonction de type « `'a list -> 'a -> 'a list` » qui supprime toutes les occurrences d'un élément `x` dans une liste `li`.
5. Écrire une fonction « `suppr_doubl1: 'a list -> 'a list` » qui supprime les doublons d'une liste en ne gardant que la dernière occurrence des doublons. Par exemple « `suppr_doubl1 [1; 2; 5; 1; 2; 3; 1; 3]` » vaut `[5; 2; 1; 3]`. Vous pouvez utiliser la fonction `List.mem` vue en cours.
6. Écrire une fonction de type « `int -> 'a -> 'a list -> 'a list` » qui insère un élément dans une liste à l'indice n.
7. Écrire une fonction de type « `'a list -> int -> ('a list * 'a list)` » qui prend en entrée une liste `li` ainsi qu'un entier `n` et renvoie deux listes `(li1, li2)` telles que `li = li1 @ li2` et `li1` est de taille `n`.

Exercice 3. Listes de listes chaînées

1. (a) Sans utiliser de fonction intermédiaire, écrire une fonction `ajout` qui prend entrée « `x: 'a` » et « `li: 'a list list` » et renvoie une copie de `li` où `x` a été ajouté au début de chaque sous-liste. Par exemple :

   `ajout 3 [[]; [2]; [2;8]]` vaut `[[3]; [3;2]; [3;2;8]]`

   (b) En déduire une fonction « `liste_prefixes: 'a list -> 'a list list` » qui renvoie la liste des préfixes de la liste en argument. Par exemple :

   « `liste_prefixes []` » vaut `[[]]`
   « `liste_prefixes [3; 2; 8]` » vaut `[[]; [3]; [3;2]; [3;2;8]]`

2. Pour tout $n \in \mathbb{N}$, on note E_n l'ensemble de toutes les listes de taille n dont chaque élément vaut `1`, `2` ou `3`. On note également L_n la liste contenant une et une seule fois chaque élément de E_n (l'ordre des sous-listes n'a pas d'importance). Par exemple L_3 vaut :

```
[[1; 1; 1]; [2; 1; 1]; [3; 1; 1]; [1; 2; 1]; [2; 2; 1]; [3; 2; 1]; [1; 3; 1];
 [2; 3; 1]; [3; 3; 1]; [1; 1; 2]; [2; 1; 2]; [3; 1; 2]; [1; 2; 2]; [2; 2; 2];
 [3; 2; 2]; [1; 3; 2]; [2; 3; 2]; [3; 3; 2]; [1; 1; 3]; [2; 1; 3]; [3; 1; 3];
 [1; 2; 3]; [2; 2; 3]; [3; 2; 3]; [1; 3; 3]; [2; 3; 3]; [3; 3; 3]]
```

   (a) Supposons que la liste L_n ait déjà été construite. Sans utiliser de fonction intermédiaire, écrire une fonction « `etape: int list list -> int list list` » qui prend en entrée L_n et renvoie L_{n+1}.

   (b) En déduire une fonction qui prend en entrée n et renvoie L_n.

Exercice 4. Facteurs et sous-mots

Soient `s1` et `s2` deux chaînes de caractères. On dit que `s1` est un ***facteur*** de `s2` s'il existe deux chaînes de caractères `s3` et `s4` telles que `s2 = s3 ^ s1 ^ s4`. Par exemple, `"abcd"` possède 11 facteurs :

`""`, `"a"`, `"b"`, `"c"`, `"d"`, `"ab"`, `"bc"`, `"cd"`, `"abc"`, `"bcd"`, `"abcd"`.

1. Écrire une fonction de type « `string -> string -> bool` » qui renvoie `true` lorsque `s1` est un facteur de `s2` et `false` sinon.
2. Même question avec des listes chaînées. Par exemple, la fonction doit renvoyer `true` pour `[2; 5; 3]` et `[10; 2; 2; 5; 3; 0; 8]`.

Soient `(s1, s2)` deux chaînes de caractères et `(n1, n2)` leurs tailles respectives. On dit que `s1` est un ***sous-mot*** de `s2` s'il existe $0 \leqslant i_0 < i_1 < \ldots < i_{n1-1} \leqslant n2 - 1$ tels que :

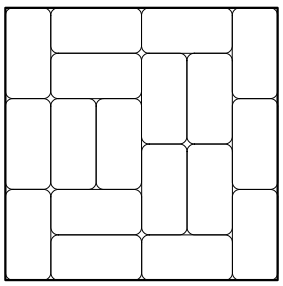
$$s1.[0] = s2.[i_0] \qquad s1.[1] = s2.[i_1] \qquad \ldots \qquad s1.[n1 - 1] = s2.[i_{n1-1}]$$

Ainsi, si `s1` est un facteur de `s2`, c'est aussi un sous-mot. Par exemple, `"abcd"` possède 16 sous-mots : ses 11 facteurs et

`"ac"`, `"ad"`, `"bd"`, `"abd"`, `"acd"`.

3. Écrire une fonction de type « `string -> string -> bool` » qui renvoie `true` lorsque `s1` est un sous-mot de `s2` et `false` sinon.
4. Même question avec des listes chaînées. Par exemple, la fonction doit renvoyer `true` pour `[10; 2; 8]` et `[10; 2; 2; 5; 3; 0; 8]`.

Exercice 5. Pavage avec des dominos



Soit $n \in \mathbb{N}$. Un ***pavage de taille*** n consiste à placer des dominos sur une grille composée de n lignes et n colonnes, en recouvrant toutes les cases sans que les dominos ne se chevauchent. Par exemple, la figure ci-contre présente un pavage de taille 6.

1. Écrire une fonction qui prend en entrée une entier n et renvoie le nombre de pavages de taille n. Pour vérifier la réponse, on pourra entrer les nombres obtenus pour les valeurs paires de n sur le site `https://oeis.org/`.

Exercice 6. Grands entiers

Contrairement à Python, le nombre d'entiers représentables par le type `int` en OCaml est fini. Dans cet exercice, on souhaite représenter des entiers de taille arbitraire en utilisant des listes chaînées. Un entier n est représenté par la liste contenant les chiffres en base 10 de n en commençant par le chiffre de poids faible. Par exemple, `5`, `100` et `1234` seront représentés par :

`[5]` `[0; 0; 1]` `[4; 3; 2; 1]`

Afin de garantir l'unicité de la représentation, on impose que le dernier élément de la liste ne soit pas `0`. On dira qu'une liste d'entiers est ***valide*** lorsque son dernier élément n'est pas `0` et que ses éléments appartiennent à {`0`,`1`,`2`,`3`,`4`,`5`,`6`,`7`,`8`,`9`}. Ainsi, la liste `[4; 3; 2; 1; 0]` n'est pas valide, en particulier elle ne représente pas `1234`. De plus, l'entier `0` sera représenté par la liste vide `[]`.

1. Écrire une fonction « `suppr_zeros: int list -> int list` » qui supprime les `0` à la fin d'une liste d'entiers.

On définit le type `nat` suivant pour représenter les entiers naturels (positifs ou nuls) :

```
type nat = int list;;
```

Dans la suite, les éléments de type `nat` seront appelés "grands entiers". On pourra supposer qu'une liste de type `nat` donnée en argument d'une fonction est valide. De plus, lorsque le type de retour d'une fonction est de type `nat`, vous devez faire en sorte que la liste renvoyée soit valide.

2. Définir une fonction « `add_nat: nat -> nat -> nat` » qui additionne deux grands entiers. Vous devez utiliser l'algorithme vu à l'école primaire pour additionner deux entiers sans convertir vers le type `int`.

Pour pouvoir utiliser la fonction `add_nat` plus facilement, on définit l'opérateur `+++` de la manière suivante :

```
let (+++) n m = add_nat n m;;
```

On peut maintenant additionner deux grands entiers avec l'expression `n +++ m`.

3. Définir un opérateur « `(---): nat -> nat -> nat` » qui prend en entrée deux grands entiers n, m, et renvoie $n - m$. Vous devez utiliser l'algorithme vu à l'école primaire pour soustraire deux entiers sans convertir vers le type `int`. Si $n < m$ votre fonction déclenchera une erreur.
4. Définir un opérateur « `( *** ): nat -> nat -> nat` » qui multiplie deux grands entiers. Vous devez utiliser l'algorithme vu à l'école primaire pour multiplier deux entiers sans convertir vers le type `int`. Pour définir cet opérateur, utilisez `( *** )` et non `(***)` sinon OCaml interprétera cette syntaxe comme un commentaire.
5. À l'aide d'une exponentiation rapide, définir un opérateur « `(^^^): nat -> int -> nat` » qui prend en entrée n, k, et qui renvoie n^k. Notez que k est de type `int`. Lorsque $n = k = 0$ ou $k < 0$, votre fonction déclenchera une erreur.
6. Écrire une fonction « `div_eucl: nat -> nat -> nat*nat` » qui prend en entrée deux grands entiers n et m, et renvoie le couple (q, r) où q et r sont le quotient et le reste dans la division euclidienne de n par m. Vous devez utiliser l'algorithme vu à l'école primaire pour effectuer la division euclidienne sans convertir vers le type `int`.

Exercices à rendre au plus tard le 20/03/2025 à 20h

Exercice 7. Permutation dans une liste

1. Écrire une fonction `perm_droite` qui effectue une permutation circulaire à droite sur une liste. Par exemple, « `perm_droite [1; 2; 3; 4; 5; 6]` » vaut `[6; 1; 2; 3; 4; 5]`.
2. Écrire une fonction `perm_gauche` qui effectue une permutation circulaire à gauche sur une liste. Par exemple, « `perm_gauche [1; 2; 3; 4; 5; 6]` » vaut `[2; 3; 4; 5; 6; 1]`.

Exercice 8. Ensemble d'entiers

Dans cet exercice, on représente un ensemble d'entiers par une liste triée sans élément en double. Par exemple, $\{1, 8, 4, -5\}$ est représenté par `[-5; 1; 4; 8]`. En particulier, on pourra supposer sans le vérifier que les fonctions demandées ci-dessous prennent en entrée des listes triées sans doublon. De plus, vous devez faire en sorte que ces fonctions renvoient des listes triées sans doublon. Toutes les fonctions doivent être écrites directement sans définir de fonction intermédiaire.

1. Écrire une fonction « `union: int list -> int list -> int list` » qui renvoie l'union des deux ensembles donnés en argument $(E_1 \cup E_2)$.
2. Écrire une fonction « `intersection: int list -> int list -> int list` » qui renvoie l'intersection des deux ensembles donnés en argument $(E_1 \cap E_2)$.
3. Écrire une fonction « `difference: int list -> int list -> int list` » qui renvoie la différence ensembliste des deux ensembles données en argument $(E_1 \setminus E_2)$.

Exercice 9. Programme auto-reproducteur (facultatif)

Considérons le programme suivant :

```
print_string "print_string ";
print_char (char_of_int 34);
print_string "print_string ";
print_char (char_of_int 34);
print_endline ";";;
```

Il affiche « `print_string "print_string ";` », c'est à dire la première ligne de son code source. On souhaiterait aller plus loin et écrire un programme qui affiche l'intégralité de son code source. Un tel programme est appelé un ***auto-reproducteur*** ou bien un ***quine***.

1. Trouver un programme auto-reproducteur (autre que le programme vide).

Exercices à rendre au plus tard le 27/03/2025 à 20h

Exercice 10. Listes chaînées

1. Écrire une fonction « `tous_uniques: 'a list -> bool` » qui renvoie `true` lorsque la liste d'entrée ne contient pas d'élément en double.
2. Écrire une fonction « `suppr_doubl2: 'a list -> 'a list` » qui supprime les doublons d'une liste en ne gardant que la première occurrence des doublons. Par exemple « `suppr_doubl2 [1; 2; 5; 1; 2; 3; 1; 3]` » vaut `[1; 2; 5; 3]`.
3. Écrire une fonction « `suppr_nth: int -> 'a list -> 'a list` » qui supprime l'élément d'indice n d'une liste.

Exercice 11. Prédicat sur une liste

Les fonctions demandées dans cet exercice prennent en argument une liste « `li: 'a list` » ainsi qu'une fonction « `pred: 'a -> bool` ». Toutes les fonctions doivent être écrites directement sans définir de fonction intermédiaire.

1. Écrire une fonction « `nth_pred : 'a list -> ('a -> bool) -> int -> 'a` » qui prend en entrée `li`, `pred` ainsi qu'un entier n et renvoie le $n^{\text{éme}}$ élément de `li` qui vérifie `pred` (dans cette question, on compte à partir de $n = 1$, pas de $n = 0$). Testez votre fonction.
2. Écrire une fonction « `keep_true : 'a list -> ('a -> bool) -> 'a list` » qui renvoie la liste composée des éléments de `li` qui vérifient `pred`.
3. Écrire une fonction « `prefixe : 'a list -> ('a -> bool) -> 'a list` » qui renvoie le plus grand préfixe de `li` dont tous les éléments vérifient `pred`.

Exercice 12. Exercice facultatif

Alice joue à un jeu de hasard : elle dispose de 10 cartes numérotées de 1 à 10, les mélange aléatoirement puis les aligne sur une table. Chacune des 10 cartes lui rapporte un point lorsque son numéro noté $k \in [\![1, 10]\!]$ est égal à sa position parmi les autres cartes. Par exemple, Alice marque 3 points avec le tirage :

`9, 1, 3, 4, 8, 2, 7, 6, 5.`

En effet, les cartes 3, 4 et 7 sont respectivement aux positions 3, 4 et 7.

1. Écrire une fonction « `nb_melanges : int -> int` » qui prend en entrée un entier $k \in \mathbb{N}$ et renvoie le nombre de mélanges pour lesquels Alice obtient k points.

Remarque. Ce qu'on a compté dans cet exercice est en fait le nombre de permutations de $[\![1, 10]\!]$ avec exactement k points fixes.