TP : Hanoi et récursivité

I. Les tours d'Hanoi

I.a. Présentation du problème

Le problème des Tours d'Hanoï consiste à faire passer des disques d'un pylône (en général celui de gauche) vers un autre (en général celui de droite).

Les règles sont simples :

- On ne déplace qu'un disque à la fois (sinon il n'y a plus de problème)
- Il est interdit de placer un disque sur un autre disque de diamètre plus petit (sur le schéma ci-dessus, le disque au milieu ne peut pas aller sur le pylône de droite)

I.b. Avec du papier

Pour vous familiariser avec le problème, vous pouvez commencer par le reproduire avec de simples bouts de papier :

- Prenez une feuille de brouillon
- Découpez-la en 4 morceaux de tailles bien différentes
- Définissez 3 emplacements sur votre table
- Empilez les 4 bouts de papier du plus large au plus petit sur l'emplacement 1
- Essayez de faire passer ces 4 bouts de papier sur l'emplacement 3 en les déplaçant un par un et sans jamais mettre un bout de papier sur un morceau plus petit

I.c. Récupération du fichier

Récupérez ensuite le fichier disponible sur le support en ligne du cours pour ce TP afin de disposer des fonctions d'affichage pour ces tours de Hanoï.

Comme vous pouvez le voir, on utilise trois références sur des listes d'entiers, nommées `tour1`, `tour2` et `tour3` pour représenter les tours d'Hanoi.

Dans chaque liste, chaque élément correspond à un disque (ce disque étant d'autant plus large que l'élément est grand)

Par exemple, pour le schéma ci-contre, on a :

- `!tour1 = [4; 5; 6; 7; 8]`
- `!tour2 = [3]`
- `!tour3 = [1; 2]`

A tout moment, vous pouvez utiliser la commande `tracer_hanoi ()` pour mettre à jour votre affichage en fonction de l'état actuel des listes `tour1`, `tour2` et `tour3`.

Vous pouvez également utiliser la fonction `initialiser_pour_les_tests ()` pour initialiser les listes conformément à l'exemple ci-dessus.

I.d. L'instruction `failwith`

L'instruction `failwith` permet à votre programme de planter. Aussi surprenant que cela puisse paraître, cela peut se révéler très utile (notamment pour ce sujet sur les tours d'Hanoi). Ainsi, quand votre programme essaye de faire quelque chose d'interdit, vous pouvez le forcer à quitter et indiquer la raison à l'utilisateur.

Pour utiliser cette instruction, il vous suffit de taper `failwith message_a_afficher`.

Imaginez par exemple une fonction `diviser` prenant en argument deux nombres entiers `a` et `b` et renvoyant le quotient de la division euclidienne de `a` par `b`, sauf dans le cas où `b` est nul, auquel cas un message d'erreur est affiché.

I.e. Enlever un disque

Ecrivez une fonction `enlever_disque` :

- prenant en argument une référence sur une liste (correspondant à une tour)
- renvoyant un entier
- telle que `enlever_disque tourX` enlève le disque situé au sommet de la tour représentée par `tourX` et retourne l'entier correspondant à ce disque
- et affichant une erreur si on tente d'enlever un disque sur une tour vide

I.f. Ajouter un disque

Ecrivez une fonction `ajouter_disque` :

- prenant en argument une référence sur une liste (correspondant à une tour) et un entier (correspondant à un disque)
- ne renvoyant rien
- telle que `ajouter_disque tourX disqueY` ajoute le disque `disqueY` au sommet de la tour `tourX`
- et affichant une erreur s'il est interdit d'ajouter ce disque sur cette tour

I.g. Déplacer un disque

Combinez les deux fonctions précédentes pour obtenir une fonction `deplacer_disque` :

- prenant en argument deux références sur des listes (correspondant à deux tours)
- ne renvoyant rien
- telle que `deplacer_disque tourX tourY` déplace le disque au sommet de la tour `tourX` vers le sommet de la `tourY`
- et affichant une erreur s'il est interdit d'effectuer un tel déplacement

I.h. Initialisation du problème

Imaginez pour finir une fonction `initialiser_tours` :

- prenant en argument un nombre entier
- ne renvoyant rien
- telle que `initialiser_tours nombre_de_disques` modifie les références `tour1`, `tour2` et `tour3` afin que
  - `tour1` contienne `[1; 2; 3; … ; nombre_de_disques]`
  - `tour2` et `tour3` soient vides

I.i. Résolution

Imaginez pour finir une fonction `résoudre`

- prenant en argument un nombre entier
- ne renvoyant rien
- telle que `résoudre nombre_de_disques` initialise le problème avec `nombre_de_disques` disques et le résout

Indice 1 : Pensez récursif !

Indice 2 : Utilisez une fonction auxiliaire prenant en argument un nombre de disques à déplacer, une tour de départ, une tour intermédiaire, et une tour d'arrivée.

II. Listes et répétitions

L'objectif de cette seconde partie est de vous faire manipuler les notions de tirage aléatoire, de listes et de récursivité.

II.a. Créer une liste aléatoire

Pour commencer, vous allez devoir créer une fonction qui génère une liste aléatoire. Vous aurez besoin pour cela de :

- la commande `Random.self_init ()`, qui permet de démarrer le générateur aléatoire (à placer et exécuter au début de cette deuxième partie)
- la fonction `Random.int` telle que `Random.int borne_max` renvoie un entier tiré au hasard entre `0` et `borne_max-1`

Utilisez cette deuxième fonction pour créer une fonction `creer_liste_simple` :

- prenant en argument un nombre entier
- renvoyant une liste d'entier
- telle que `creer_liste_simple nombre_elements` renvoie une nouvelle liste composés de `nombre_elements` tirés au hasard entre 0 et 100.

II.b. Créer une liste aléatoire sans doublons

On souhaite désormais que la liste générée ne possède aucun doublon (un même élément ne peut donc pas apparaître deux fois dans cette liste)

Utilisez cette deuxième fonction pour créer une fonction `creer_liste` :

- prenant en argument un nombre entier (supposé inférieur à 100)
- renvoyant une liste d'entier
- telle que `creer_liste nombre_elements` renvoie une nouvelle liste composés de `nombre_elements` tirés au hasard entre 0 et 100, et deux à deux distincts

Indice : vous pouvez utiliser la fonction `List.mem` pour vérifier si un élément appartient à une liste (`List.mem x l` renvoie `true` si `x` apparait dans `l`, et false sinon)

Remarque : Pourquoi ne faut-il pas lancer cette fonction avec `nombre_elements > 101` ?

II.c. Ajouter des doublons

Imaginez ensuite une fonction `doublonner_liste` :

- prenant en argument une liste d'entier (supposée sans doublons)
- renvoyant une nouvelle liste
- telle que `doublonner_liste l` renvoie une nouvelle liste composée des éléments de `l`, chacun étant répété deux fois (exemple : `[2; 3]` devient `[2; 2; 3; 3]`)

II.d. Ajouter plus de doublons

Imaginez ensuite une fonction `doublonner_liste_mieux` :

- prenant en argument une liste d'entier (supposée sans doublons)
- renvoyant une nouvelle liste
- telle que `doublonner_liste_mieux l` renvoie une nouvelle liste `l2` composée des éléments de `l`, le $i^{ème}$ élément de `l` étant répété `i` fois dans `l2` (exemple : `[2; 3; 1]` devient `[2; 3; 3; 1; 1; 1]`)

II.e. Supprimer les doublons

Imaginez pour finir une fonction `supprimer_doublons` :

- prenant en argument une liste d'entier (supposée avec doublons)
- renvoyant une nouvelle liste
- telle que `supprimer_doublons l` renvoie une nouvelle liste `l2` composée des éléments de `l`, chaque élément de l2 ne pouvant être identique à son successeur. (exemple : `[2; 2; 3; 3; 3; 1]` devient `[2; 3; 1]`)