Partie I : structure "trouver-unir"

Une structure "Union-Find" doit permettre notamment de manipuler des partitions d'ensemble. On représentera une telle partition $\{E_1, \cdots, E_p\}$ d'un ensemble $[\![0, n-1]\!]$ par un tableau `t` de taille $n-1$d'entiers tel que `t.(i)` est l'entier j tel que $i \in E_j$
Par exemple, `[| 0; 0; 1; 2; 1|]` représentation la partition $\{\{0,1\},\{2,4\},\{3\}\}$ de l'ensemble $[\![0,4]\!]$.
Une telle structure devra être munie de fonctions "trouve" (ou "find") et "unit" (ou "union") permettant de trouver le sous-ensemble auquel appartient un élément et de modifier la partition en fusionnant deux sous-ensembles.

Question 0
Ecrire une fonction `partition : int -> int array` telle que cette fonction appelée sur un entier n crée la partition $\{\{0\}, \cdots, \{n-1\}\}$.

Question 1
Ecrire une fonction `trouver : int array -> int -> int` telle que `trouve t i` renvoie l'entier j tel que $i \in E_j$ où `t` représente la partition $\{E_j \,/\, 0 \leq j \leq p-1\}$.

Question 2
Ecrire une fonction `unir : int array -> int -> int -> unit` telle que `unit t i j` modifie le tableau `t` représentant une partition $\{E_j \,/\, 0 \leq j \leq p-1\}$ en fusionnant E_k et E_l, les sous-ensembles contenant `i` et `j`. On supposera que `i` et `j` sont dans deux sous-ensembles distincts.

Question 3
Donner les complexités de ces fonctions.

Pour améliorer ces dernières, on représentera un sous-ensemble E_j par un arbre enraciné donc une partition par une forêt d'arbres enracinés (une union d'arbre). Une telle forêt sera représentée par un tableau `t` de taille $n-1$.
Si $0 \leq i \leq n-1$, `t.(i)` sera le père de `i` s'il en a un, `i` si `i` est la racine de l'arbre contenant les éléments de E_j.
Par exemple, `[| 0; 0; 0; 1; 4; 1; 4|]` représente la forêt à deux arbres :
`N(0,N(1,N(3,Vide,Vide),N(5,Vide,Vide)),N(2,Vide,Vide))` et `N(4,N(6,Vide,Vide),Vide)` qui représentent la partition $\{\{0,1,3,5,2\},\{4,6\}\}$.
Pour trouver l'ensemble E_j tel que $i \in E_j$, on renverra la racine de l'arbre représentant E_j ; pour unir deux sous-ensembles E_j et E_i, on choisira l'une des deux racines représentant E_i et E_j pour être racine de l'arbre fusionné et on lui ajoutera un sous-arbre enraciné en l'autre racine.
Dans l'exemple précédent, pour fusionner $\{0,1,3,5,2\}$ et $\{4,6\}$, il suffira d'écrire `t.(4)<-0;`.

Question 4
Ecrire des fonctions `trouver2 : int array -> int -> int` et `unir2 : int array -> int -> int -> unit` telles que `trouver2 t i` renvoie la racine de l'arbre dans lequel se trouve `i` et `unir2 t i j` modifie `t` de sorte que l'une des racines des arbres de `i` et `j` devienne le père de l'autre (on supposera que ces deux entiers ne figurent pas initialement dans le même arbre).

Question 5
Créer une fonction `hauteur : int arrayt -> int -> int` renvoyant la hauteur de l'arbre auquel un élément appartient.

Question 6
Ecrire une fonction `unir3` qui réalise l'union en choisissant pour racine du nouvel arbre celle de l'arbre de plus grande hauteur.

Partie II : tri par tas de couples

On considère dans cette partie des tas-min stockant des couples de type `'a*int` qui seront rangés par ordre croissant selon le second élément. On rappelle qu'un tas de taille n est représenté le type
`type 'a tas = { mutable taille : int ; contenu : 'a*int array}`

Question 7
Ecrire les fonctions `insere : 'a*int -> 'a tas -> unit` et `retire : 'a tas -> 'a` vues en cours telles que la première modifie un tas en insérant un élément et sa priorité dans le tas et la deuxième enlève et renvoie le premier élément du couple prioritaire d'un tas.

Partie III : algorithme de Kruskal

Question 8
Ecrire une fonction `kruskal : int list array -> int*int list` renvoyant la liste des arêtes (représentées par un couple de leurs extrémités) d'un arbre couvrant de poids minimal.

Partie IV : tri topologique

Etant donné un graphe orienté acyclique, la relation $i \geq j$ définie par le fait qu'il existe un chemin de i vers j est une relation d'ordre (exercice : le vérifier). En général, elle n'est pas totale (par exemple, si `g= [| [2] ; [2]; [] |]`, ni 1, ni 0 n'est supérieur à l'autre).
Un tri topologique sur un tel graphe est une extension de cette relation en une relation d'ordre totale. Sur l'exemple précédent, $0 \geq 1 \geq 2$ ou $1 \geq 0 \geq 2$ sont des tris topologiques. De tels tris seront représentés par des listes (en l'espèce, les listes `[0;1;2]` et `[1;0;2]`.

Question 9
Ecrire une fonction `parcours : int list array -> int -> int list` qui lance un parcours de graphe à partir d'un sommet et renvoie la liste des sommets parcourus dans l'ordre de parcours.

Question 10
Ecrire une fonction `tri_topo : int list array -> int list` qui renvoie un tri topologique d'un graphe orienté acyclique.

Remarque : un graphe orienté acyclique peut représenter un ensemble de tâches (les sommets) dont certaines dépendent d'autres (la dépendance étant représentée par une arête) ; un tri topologique représente dans ce cas un ordre d'exécution admissible de la suite des tâches à effectuer.

Partie V : graphes bipartis

Question 11
Ecrire une fonction `biparti : int list array -> bool` vérifiant si un graphe est biparti (ou bicoloriable).

Question 12
Modifier la fonction précédente en une fonction `bipartion : int list array -> int array` renvoyant un tableau de 0 et 1 qui soit une bipartition du graphe en argument (supposé biparti).

Etant donnée une bipartion (X, Y) d'un graphe bipartie $G = (S, A)$ (c'est-à-dire que $S = X \cup Y$ et tout élément de X n'a que des voisins des Y et réciproquement).
Un couplage sera représenté par deux tableaux `x` et `y` tel que `x.(i)` sera `-2` si `i` n'est pas dans X, sera `-1` s'il n'est pas apparié et sera le sommet auquel il est apparié sinon. Le tableau `y` obéira aux même règles.

`Question 13`
Ecrire une fonction `chemin_augmentant : int array -> int array -> (int array*int)` qui, étant donné un couplage représenté par des tableaux `x`, `y`, effectue un parcours en largeur depuis les sommets libres de X affectant à chaque sommet découvert le sommet depuis lequel il a été découvert et qui s'arrêtera dès la découverte d'un sommet

libre de Y. La fonction renverra le tableau des successeurs et le premier sommet libre découvert ; s'il n'y en a pas, la fonction renverra en deuxième argument -1.

Question 14
Ecrire une fonction `difference_symetrique : int -> int array -> int array -> int array ->unit` telle que `difference_symetrique i t x y` modifie le couplage M représenté par `x`, `y` à l'aide du chemin augmentant C s'achevant en `i` représenté par `t` en un couplage augmenté $M\Delta C$.

Question 15
Ecrire une fonction `couplage_maximum : int list array -> int array*int array` renvoyant un couplage maximum d'un graphe biparti.