Un même identificateur écrit dans deux polices de caractères différentes désignera la même entité, mais du point de vue mathématique pour la police en italique (par exemple n) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`).

Le World Wide Web, ou Web, est un ensemble de pages Web (identifiées de manière unique par leurs adresses Web, ou URL pour Uniform Resource Locators, de la forme http ://mines-ponts.fr/index.php) reliées les unes aux autres par des hyperliens. Le Web est souvent modélisé comme un graphe orienté dont les sommets sont les pages Web et les arcs les hyperliens entre pages. Le Web étant potentiellement infini, on s'intéresse à des sous-graphes du Web obtenus en naviguant sur le Web, c'est-à-dire en le parcourant page par page, en suivant les hyperliens d'une manière bien déterminée. Ce parcours du Web pour en collecter des sous-graphes est réalisé de manière automatique par des logiciels autonomes appelés Web crawlers ou crawlers en anglais, ou collecteurs en français.

Fonctions utilitaires

Nous allons tout d'abord coder certaines fonctions de manipulation de structures de données de base, qui seront utiles dans le reste de l'exercice.

1) Coder une fonction `aplatir : ('a * 'a list) list → 'a list`, telle que, si `liste` est une liste de couples $[(x_1, l_{x_1}); \ldots; (x_n, l_{x_n})]$, où chaque x_i est un élément de type `'a`, et l_{x_i} une liste d'éléments de type `'a` de la forme $[y_{i1}; \ldots; y_{ik_i}]$, `aplatir liste` est une liste d'éléments de type `'a`

$$[x_1; y_{11}; \ldots; y_{1k_1}; x_2; y_{21}; \ldots; y_{2k_2}; \ldots; x_n; y_{n1}; \ldots; y_{nk_n}]$$

   On pourra utiliser l'opération de concaténation des listes `@`.

2) Coder une fonction `tri_fusion : (a * b) list → (a * b) list` triant une liste de couples (x, y) par ordre décroissant de la valeur de la seconde composante y de chaque couple. On devra utiliser l'algorithme de tri par partition-fusion (aussi appelé « tri fusion »). Quelle est la complexité de cet algorithme ?

On va utiliser dans la suite de l'exercice un type de données `dictionnaire` qui permet de stocker des couples formés d'une chaîne de caractères (une clef) et d'un entier (une valeur). On dit que le dictionnaire associe la valeur à la clef. À chaque clef présente dans le dictionnaire est associée une seule valeur. Les fonctions que l'on veut définir sont :

- `dictionnaire_vide : unit → dictionnaire`.
  L'appel `dictionnaire_vide ()` crée un nouveau dictionnaire vide.
- `ajoute : string → int → dictionnaire → dictionnaire`.
  L'appel `ajoute clef valeur dict` renvoie un nouveau dictionnaire identique au dictionnaire *dict*, sauf qu'un couple $(clef, valeur)$ y a été ajouté. Cette fonction s'exécute en temps $O(\log n)$ où n est le nombre d'entrées du dictionnaire.
  On choisit, par convention, d'ajouter le couple $(clef, valeur)$ même s'il y a déjà un couple avec la même clef.
- `contient : string → dictionnaire → bool`.
  L'appel `contient clef dict` renvoie un booléen indiquant s'il y a un couple dont la clef est $clef$ dans le dictionnaire *dict*. Cette fonction s'exécute en temps $O(\log n)$ où n est le nombre d'entrées du dictionnaire.
- `valeur : string → dictionnaire → int`.
  L'appel `valeur clef dict` renvoie la valeur associée à la clef $clef$ dans le dictionnaire *dict*. Cette fonction s'exécute en temps $O(\log n)$ où n est le nombre d'entrées du dictionnaire. Cette fonction ne peut être appelée que si la clef $clef$ est présente dans le dictionnaire.

On va implémenter les dictionnaires à l'aide d'un arbre binaire de recherche. On définit donc le type

```
type dictionnaire =
  |Vide
  |N of dictionnaire*(sting*int)*dictionnaire;;
```

3) Implémenter les fonctions `dictionnaire_vide`, `ajoute`, `contient` et `valeur`.

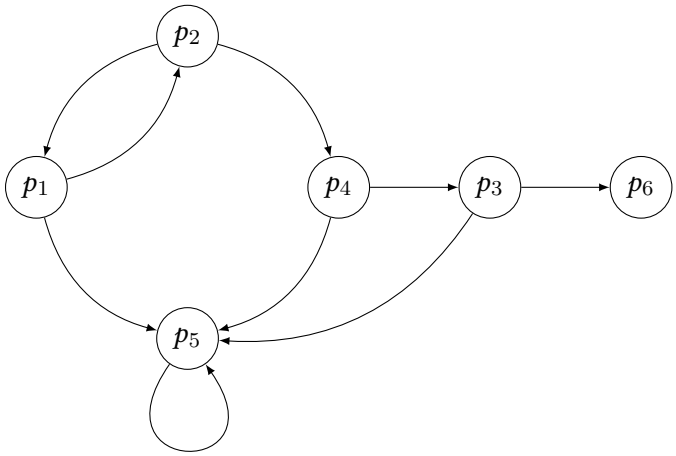
4) Coder la fonction `unique : string list → string list * dictionnaire`, qui est telle que `unique liste` renvoie un couple (*liste′*, *dict*) où *liste′* est la liste des chaînes de caractères de listes distinctes (dans l'ordre de leur première occurrence dans `liste`) et où *dict* associe à chaque chaîne de caractères dans *liste′* sa position dans *liste′* (en numérotant à partir de 0). Ainsi l'appel `unique ["x";"zz";"x";"x";"zz";"yt"]` renvoie un couple formé de la liste `["x";"zz";"yt"]` et d'un dictionnaire associant à `"x"` la valeur `0`, à `"zz"` la valeur `1` et à `"yt"` la valeur `2`.
5) Quelle est la complexité de la fonction `unique` en terme de la longueur n de la liste *liste* en argument et du nombre m d'éléments distincts dans la liste *liste* ? Justifier la réponse.

Crawler simple

Nous allons maintenant implémenter un crawler simple en Caml.

On suppose fournie une fonction `recupere_liens : string → string list` prenant en argument l'URL d'une page Web p et renvoyant la liste des URL des pages q pour lesquelles il existe un hyperlien de p à q, dans l'ordre lexicographique.

Pour illustrer le comportement de cette fonction, nous considérons un exemple de mini-graphe du Web à six pages et neuf hyperliens comme suit :



Dans cette représentation, p_1, p_2, etc., sont les URL de pages Web (simplifiées pour l'exemple), et les arcs représentent les hyperliens entre pages Web.

Dans ce mini-graphe, un appel à `recupere_liens "p1"` retourne la liste `["p2"; "p5"]`.

Un crawler est un programme qui, à partir d'une URL, parcourt le graphe du Web en visitant progressivement les pages dont les liens sont présents dans chaque page rencontrée, en suivant une stratégie de parcours de graphe (par exemple, largeur d'abord, ou profondeur d'abord). À chaque nouvelle page, si celle-ci n'a pas déjà été visitée, tous ses hyperliens sont récupérés et ajoutés à une liste de liens à traiter. Le processus s'arrête quand une condition est atteinte (par exemple, un nombre fixé de pages ont été visitées). Le résultat renvoyé par le crawler, que l'on définira plus précisément plus loin, est appelé un crawl.

6) Coder la fonction `recupere_liens : string → string list` qui correspond au graphe ci-dessus.
7) Coder `crawler_bfs : int → string → (string * string list) list` qui prend en entrée un nombre n de pages et une URL u et renvoie en sortie une liste de longueur au plus n de couples (v, l) où v est l'URL d'une page visitée (les pages apparaissant dans l'ordre où elles ont été visitées) et l la liste des liens récupérés sur la page v. On demande que `crawler_bfs` parcourt le graphe du Web en suivant une stratégie en largeur d'abord (breadth-first search), c'est-à-dire en visitant en priorité les pages rencontrées le plus tôt dans l'exploration. Le crawler doit visiter n pages distinctes, et donc appeler n fois la fonction `recupere_liens` (sauf

s'il n'y a plus de pages à visiter). On utilisera une variable de type dictionnaire pour se souvenir des pages déjà visitées.

Par exemple, sur le mini-graphe, `crawler_bfs 4 "p1"` pourra renvoyer le résultat :

```
["p1", ["p2"; "p5"];
 "p2", ["p1"; "p4"];
 "p5", ["p5"];
 "p4", ["p3"; "p5"]]
```

8) Coder `crawler_dfs : int → string → (string * string list) list` qui prend en entrée un nombre n de pages et une URL u et renvoie en sortie une liste de longueur au plus n de couples (v, l) où v est l'URL d'une page visitée (les pages apparaissant dans l'ordre où elles ont été visitées) et l la liste des liens récupérés sur la page v. On demande que `crawler_dfs` parcourt le graphe du Web en suivant une stratégie en profondeur d'abord (depth-first search), c'est-à-dire en visitant en priorité les pages rencontrées le plus récemment dans l'exploration. Le crawler doit visiter n pages distinctes, et donc appeler n fois la fonction `recupere_liens` (sauf s'il n'y a plus de pages à visiter). On utilisera une variable de type dictionnaire pour se souvenir des pages déjà visitées.

Par exemple, sur le mini-graphe, crawler_dfs 4 "p1" pourra renvoyer le résultat :

```
["p1", ["p2"; "p5"];
 "p2", ["p1"; "p4"];
 "p4", ["p3"; "p5"];
 "p3", ["p5"; "p6"]]
```

9) Coder une fonction Caml `construit_graphe : (string * string list) list → string list * int array array` telle que si *crawl* est le résultat renvoyé par un crawler (une liste de couples formés d'une URL v et de la liste des liens récupérés sur la page v), alors `construit_graphe crawl` est un couple (l, G) où l est une liste de toutes les URL de pages contenues dans la liste *crawl* et G est la matrice d'adjacence du sous-graphe partiel du Web restreint aux pages de la liste l : G_{ij} est le nombre de liens découverts dans le crawl de la page d'indice i dans l vers la page d'indice j dans l. On fera commencer les indices à 0. Pour coder la fonction `construit_graphe`, on pourra utiliser les fonctions `aplatir` et `unique`.

Par exemple, sur le mini-graphe, si crawl est une variable contenant le résultat de l'appel `crawler_bfs 4 "p1"` (voir question 7), alors `construit_graphe crawl` doit renvoyer :

```
["p1"; "p2"; "p5"; "p4"; "p3"],
[|[|0; 1; 1; 0; 0|];
  [|1; 0; 0; 1; 0|];
  [|0; 0; 1; 0; 0|];
  [|0; 0; 1; 0; 1|];
  [|0; 0; 0; 0; 0|]|]
```

En particulier :
- p_3 apparaît même s'il n'a pas été visité dans le crawl ;
- p_6 n'apparaît pas car il n'a pas été découvert dans le crawl ;
- l'hyperlien de p_3 à p_5 n'apparaît pas car p_3 n'a pas été visité.

Calcul de PageRank

PageRank est une manière d'affecter un score à l'ensemble des pages du Web, imaginée par Sergey Brin et Larry Page, les fondateurs du moteur de recherche Google. L'introduction de PageRank a révolutionné la technologie des moteurs de recherche sur le Web. Nous allons maintenant implémenter le calcul de PageRank.

Étant donnée une partie du Web (où l'ensemble des pages est indexé entre 0 et $n-1$), la matrice de surf aléatoire dans cette partie du Web est la matrice M de taille $n \times n$ définie comme suit :

- S'il n'y a aucun lien depuis une page Web d'indice i, alors pour tout j, $M_{ij} := 1/n$.
- Sinon, s'il y a k_i liens depuis la page Web d'indice i, alors pour tout j, on a $M_{ij} := (1-d) \times G_{ij}/k_i + d/n$, où G_{ij} est le nombre de liens depuis la page d'indice i vers la page d'indice j et d est un nombre réel fixé appartenant à $[0, 1]$ (on prend souvent $d = 0,15$).

Cette matrice peut être vue comme décrivant la marche aléatoire d'un surfeur sur le Web. À chaque fois que celui-ci visite une page Web :

- Si cette page ne comporte aucun lien, il visite une page Web arbitraire, choisie aléatoirement de façon uniforme.
- Si cette page comporte au moins un lien, il visite avec une probabilité égale à $1 - d$ un des liens sortants de cette page, et avec une probabilité égale à d une page Web arbitraire, choisie aléatoirement de façon uniforme.

10) Coder `surf_aleatoire : float → int array array → float array array` telle que si d est un nombre entre 0 et 1, et si G est la matrice d'adjacence d'un sous-graphe partiel du Web, alors `surf_aleatoire d G` renvoie la matrice M de surf aléatoire dans ce sous-graphe.

11) Coder `multiplie : float array → float array array → float array`, une fonction prenant en argument un vecteur ligne v de taille n et une matrice M de taille $n \times n$ et renvoyant le vecteur ligne w de taille n résultant du produit de v par la matrice : $w = vM$.

Le PageRank des pages d'un sous-graphe du Web à n pages se calcule par des multiplications successives d'un vecteur ligne par la matrice de surf aléatoire M de ce sous-graphe. Plus précisément, soit θ un nombre réel strictement positif (par exemple, $\theta = 10^{-4}$) et soit $v^{(0)}$ le vecteur ligne de taille n dont toutes les composantes valent $1/n$. On pose pour un entier naturel p arbitraire $v^{(p)} := v^{(0)}M^p$. L'algorithme de PageRank calcule la suite des $v^{(p)}$ pour $p = 0, 1, \ldots$ jusqu à ce que $\|v^{(p+1)} - v^{(p)}\|_1 \leq \theta$ et renvoie alors le vecteur $v^{(p+1)}$, considéré comme le vecteur des scores de PageRank. On peut montrer (à l'aide du théorème de Perron-Frobenius) que l'algorithme termine dès lors que d est strictement positif.

PageRank est utilisé pour affecter un score d'importance aux pages du Web. Le vecteur de scores v retourné par l'algorithme de PageRank donne dans v_i le score d'importance de la page d'indice i. Les pages de plus haut score de PageRank sont considérées comme les plus importantes.

12) Coder `pagerank : float → float array array → float array`, une fonction prenant en argument un nombre $\theta > 0$ et une matrice M de surf aléatoire d'un sous-graphe du Web et renvoyant le vecteur des scores de PageRank pour θ et M. La fonction `pagerank` devra faire appel à la fonction `multiplie` précédemment codée.

13) Coder

`calcule_pr : float → float → (string * string list) list → (string * float) list`

telle que `calcule_pr d theta crawl` renvoie une liste de couples (u, s), un couple pour chaque URL découverte dans le crawl, triée par valeur décroissante de s, où u est l'URL de cette page et s son score de PageRank. Ici, d et θ sont les deux paramètres nécessaires au calcul de la matrice de surf aléatoire et du PageRank respectivement. On pourra faire appel à la fonction `tri_fusion` et à l'ensemble des fonctions développées dans les questions précédentes.