IN 102
11 janvier 2008

Contrôle

ENSTA ParisTech

NOM : **Prénom :**

- tous les documents (poly, slides, TDs, livres...) sont autorisés.
- les réponses doivent toutes tenir dans les cases prévues dans l'énoncé (pas de feuilles supplémentaires).
- le code des algorithmes/fonctions pourra être donné soit en C/C++ soit en pseudo-code. En C la syntaxe n'a pas besoin d'être exacte, mais le code doit être présenté clairement (accolades, indentation, écriture lisible...). En pseudo-code, aucune syntaxe n'est imposée, mais tout devra être suffisamment détaillé pour ne pas laisser de place aux ambiguïtés (écrire "parcourir les sommets du graphe G" n'est pas assez précis, mais "parcourir les voisins du sommet S" est correct).
- le barème actuel de ce contrôle aboutit à une note sur 130 points. La note finale sur 20 sera dérivée de la note sur 130 grâce à une fonction strictement croissante qui n'est pas encore définie (cela ne sert à rien de la demander).
- n'oubliez pas de remplir votre nom et votre prénom juste au dessus de ce cadre.

Exercice 1 : QCM (10 points)

Chaque bonne réponse rapporte 1 point. Chaque mauvaise réponse enlève 1 point. Ne répondez pas au hasard, la note totale peut être négative !

1.a] Chassez l'intrus parmi ces algorithmes de tri :

☐ tri fusion, ☐ tri rapide, ☐ tri par insertion, ☐ tri par tas.

1.b] On insère les éléments 4, 3, 12, 7, 9 (dans cet ordre) dans une *file*. Dans quel ordre vont-ils ressortir ?

☐ 9, 7, 12, 3, 4 ☐ 3, 4, 7, 9, 12 ☐ 4, 3, 12, 7, 9 ☐ 12, 9, 7, 4, 3.

1.c] On insère les éléments 4, 3, 12, 7, 9 (dans cet ordre) dans une *pile*. Dans quel ordre vont-ils ressortir ?

☐ 9, 7, 12, 3, 4 ☐ 3, 4, 7, 9, 12 ☐ 4, 3, 12, 7, 9 ☐ 12, 9, 7, 4, 3.

1.d] On insère les éléments 4, 3, 12, 7, 9 (dans cet ordre) dans un *tas*. Dans quel ordre vont-ils ressortir ?

☐ 9, 7, 12, 3, 4 ☐ 3, 4, 7, 9, 12 ☐ 4, 3, 12, 7, 9 ☐ 12, 9, 7, 4, 3.

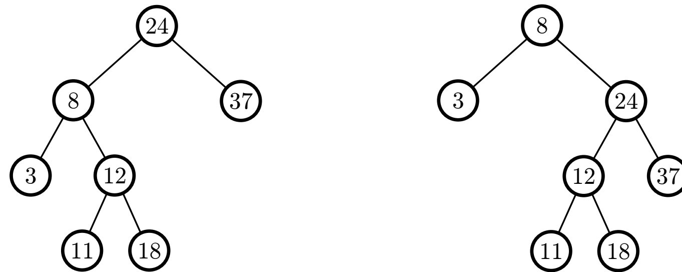
1.e] À laquelle des structures suivantes s'apparente le plus une représentation de graphe par listes de successeurs ?

☐ une pile, ☐ un arbre binaire,
☐ une table de hachage, ☐ un tableau bidimensionnel.

1.f] Quelle est la complexité *dans le pire cas* de la recherche d'un élément dans un arbre binaire de recherche de hauteur h contenant n nœuds ?

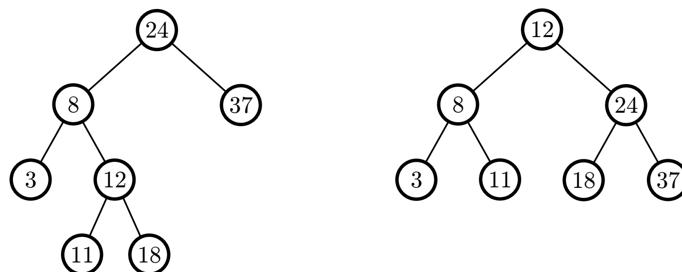
☐ $\Theta(n)$ ☐ $\Theta(\log n)$ ☐ $\Theta(h)$ ☐ $\Theta(\log h)$

1.g] Quelle transformation transforme l'arbre de gauche en celui de droite ?



☐ une rotation droite,
☐ une rotation gauche,
☐ une double rotation,
☐ aucun des trois.

1.h] Quelle transformation transforme l'arbre de gauche en celui de droite ?

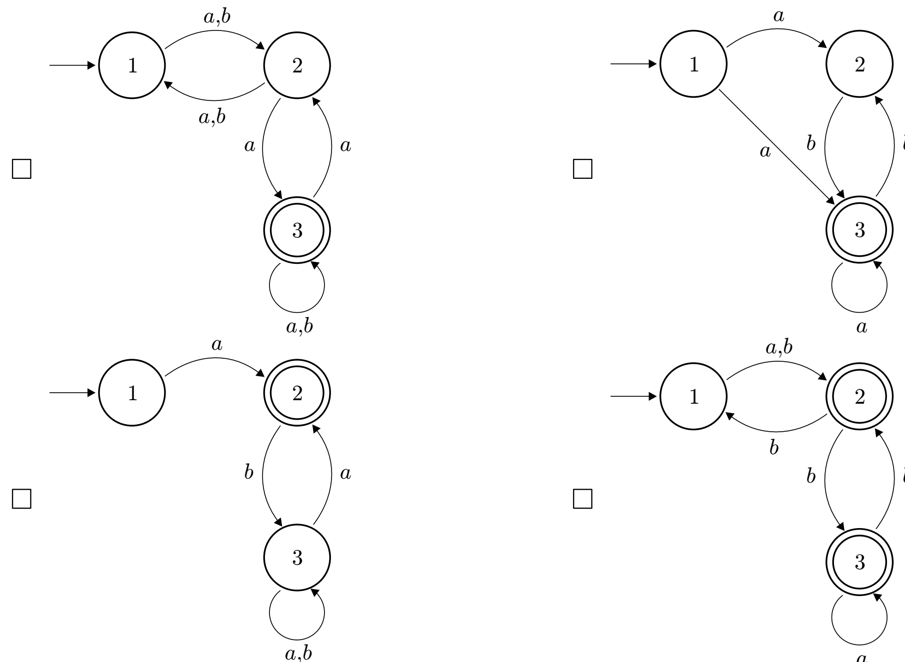


☐ une rotation droite,
☐ une rotation gauche,
☐ une double rotation,
☐ aucun des trois.

1.i] Chassez l'intrus parmi ces langages reconnus par un automate :

☐ $(a|b)^*$ ☐ $(a^*|b^*)$ ☐ $(a|b^*)^*$ ☐ $(a^*|b)^*$

1.j] Lequel de ces automates non-déterministes *ne reconnaît pas* le mot *abba* ?



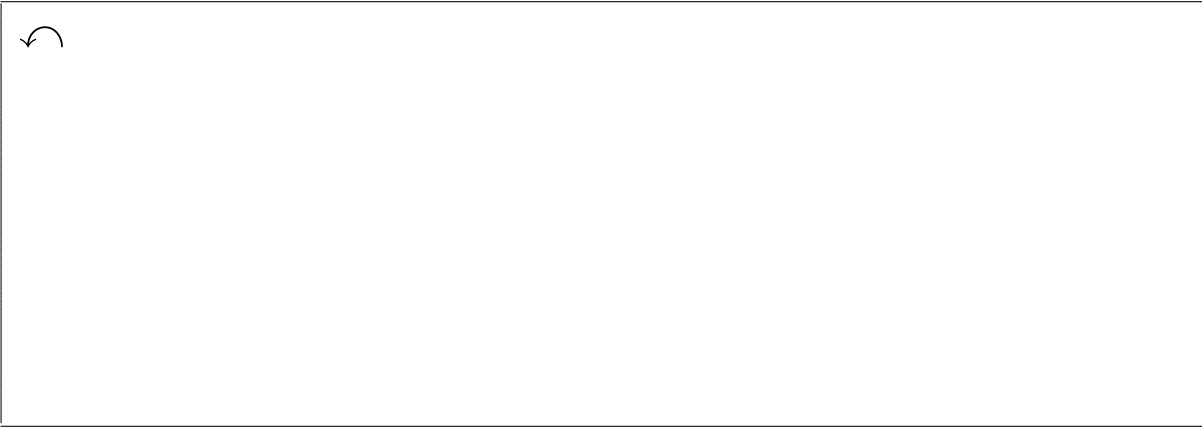
Exercice 2 : Implémentation d'une pile et d'une file avec un tableau (12 points)

Nous avons vu en cours comment implémenter une file ou une pile à l'aide d'une liste chaînée. Cependant il est aussi possible de les programmer à l'aide d'un tableau. Pour cela on peut utiliser les deux structures suivantes :

```
1  struct pile {
2    int n;
3    int max;
4    int* tab;
5  };
6
7  struct file {
8    int n;
9    int p;
10   int max;
11   int* tab;
12 };
```

Pour ces deux structures, `tab` est un tableau de taille `max` et `n` correspond au nombre d'éléments présents dans la structure. Pour la file, on utilise en plus un indice `p` qui est la position à laquelle le prochain élément doit être inséré et on remplit le tableau de façon cyclique : quand on arrive à la fin du tableau on recommence du début.

(6 pts) **2.a]** Écrivez les fonction `push_pile`, `pop_pile` et `pile_is_empty` qui vont respectivement ajouter un élément à la pile, en retirer un et tester si la pile est vide. On choisit la convention d'écrire un message d'erreur en cas de débordement de la pile.



(6 pts) **2.b]** De même écrivez les fonctions `push_file`, `pop_file` et `file_is_empty`. Encore une fois, en cas de débordement, affichez un message d'erreur.

Exercice 3 : Une mauvaise implémentation de tas (30 points)

Sur les conseils d'une de ses collègues, un programmeur en charge d'implémenter une file de priorité décide d'utiliser un tas. Malheureusement, ce programmeur n'ayant jamais suivi de cours d'algorithmique, il décide d'implémenter ce tas comme un arbre binaire en utilisant la structure suivante :

```
1 struct heap {
2   int val;
3   heap* left;
4   heap* right;
5 };
```

Pour programmer les opérations d'insertion/suppression, il se rend vite compte qu'il faut tout d'abord être capable de compter le nombre d'éléments présents dans un tas. Nous supposons donc que le tas contient n éléments, et l'on souhaite pouvoir déterminer la valeur de n.

(5 pts) **3.a**] En vous inspirant d'un parcours d'arbre en profondeur, écrivez une fonction récursive `int rec_count(heap* H)` qui compte le nombre d'éléments dans un tas (rappel : vous pouvez écrire cette fonction soit en C/C++, soit en pseudo-code). Quelle est la complexité de cet algorithme ?

Ce premier algorithme n'exploite pas l'une des propriétés fondamentales d'un tas : c'est un arbre binaire complet. Pour exploiter cette propriété, le programmeur décide de programmer les deux fonctions suivantes :

```
1 int LD (heap* H) {
2   if (H == NULL) {
3     return 0;
4   } else {
5     return 1 + LD(H->left);
6   }
7 }
```

```
1 int RD (heap* H) {
2   if (H == NULL) {
3     return 0;
4   } else {
5     return 1 + RD(H->right);
6   }
7 }
```

(3 pts) **3.b]** Que font les fonctions `LD` et `RD` ?

(3 pts) **3.c]** Si `LD(H)` retourne h, quelles valeurs peut retourner `RD(H)` ? Qu'est-ce que cela signifie quand `LD(H)` et `RD(H)` retournent la même valeur et combien d'éléments contient alors `H` ?

(4 pts) **3.d]** Si `LD(H)` et `RD(H)` retournent des valeurs différentes, le programmeur décide de regarder les valeurs retournées par `RD(H->left)` et `LD(H->right)`. Selon les égalités/différences entre `LD(H)`, `RD(H->left)`, `LD(H->right)` et `RD(H)`, donnez un encadrement du nombre d'éléments dans le tas `H`.

(6 pts) **3.e]** Écrivez un algorithme récursif *efficace* `int count(heap* H)` qui utilise les fonctions `LD` et `RD` pour compter les éléments dans le tas (note : pour ceux qui programment en C, 2^i se calcule efficacement avec la commande `(1 << i)`). Quelle est sa complexité en fonction de n ?

Maintenant qu'il sait compter efficacement le nombre d'éléments dans un tas, notre programmeur veut pouvoir trouver facilement le “dernier” élément du tas pour programmer une suppression.

(6 pts) **3.f]** Écrivez un algorithme efficace `heap* last(int n)` qui retourne un pointeur sur le n-ième élément du tas (on suppose que cet élément existe).

Afin d'optimiser son implémentation, notre programmeur décide maintenant de stocker la valeur de n dans une variable. Ainsi, à chaque insertion/suppression il n'a plus besoin de recalculer cette valeur et doit simplement l'incrémenter ou la décrémenter.

(3 pts) **3.g**] Quel est alors le coût d'une insertion ou d'une suppression dans un tel tas de n éléments ? En particulier, avec cette implémentation, quel est le coût d'un tri par tas ?

Exercice 4 : Graphes orientés (25 points)

On se donne les structures de données suivantes :

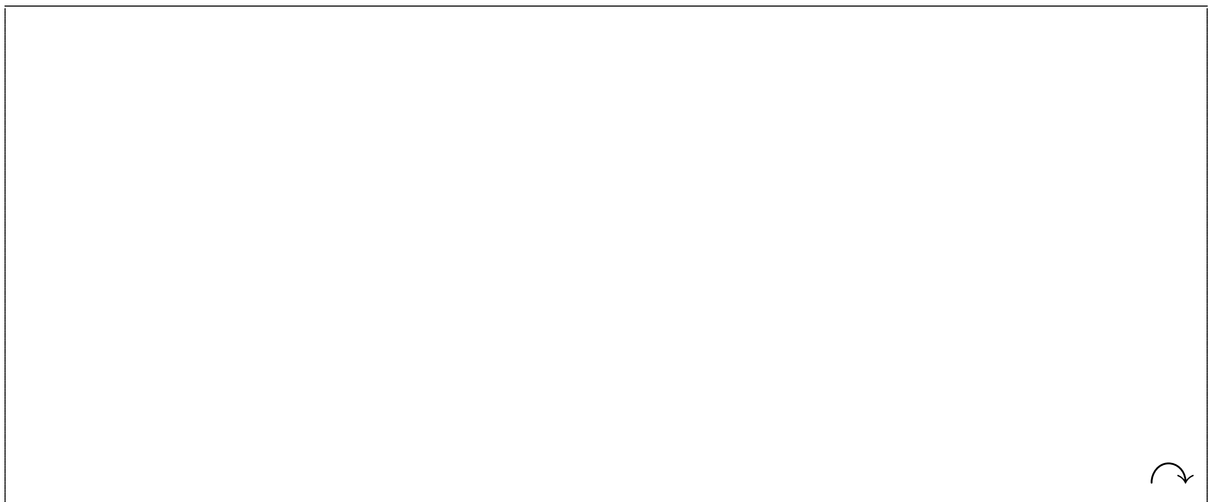
```
1 int** adj_mat;
2
3 struct vertex_list {
4   int vertex;
5   vetex_list* next;
6 };
7 vertex_list** successor_list;
```

Le tableau `adj_mat` correspond à la matrice d'adjacence d'un graphe et `successor_list` au tableau de listes de la représentation par liste de successeurs. On suppose que le graphe a `n` sommets et `A` arcs.

(6 pts) **4.a**] Écrivez deux fonctions qui convertissent un graphe d'une représentation vers l'autre et réciproquement. Quelles sont leurs complexités ?

(6 pts) **4.b]** Étant donné un sommet `v` du graphe, on cherche à construire la liste des sommets `u` du graphe tels qu'il existe un arc allant de `u` à `v`. Écrivez deux fonctions, l'une utilisant la représentation par matrice d'adjacence, l'autre par liste de successeurs qui permettent de construire cette liste de sommets. On veut que ces fonctions aient une complexité linéaire en la taille du graphe.

(7 pts) **4.c]** On veut maintenant itérer les fonctions précédentes pour déterminer l'ensemble des antécédents de `v`, c'est-à-dire, l'ensemble des sommets `u` du graphe tels qu'il existe un *chemin* allant de `u` à `v`. Quelle solution proposez vous pour gérer efficacement les éventuels cycles du graphe qui pourraient faire boucler indéfiniment un tel algorithme ? Écrivez une fonction qui utilise l'une des deux fonctions de la question précédente (celle de votre choix) et qui construit cette liste d'antécédents. Quelle est sa complexité ?



(6 pts) **4.d]** Une technique plus efficace pour trouver tous les antécédents d'un sommet commence par d'abord créer un nouveau graphe, identique au précédent, mais dont tous les arcs ont été inversés. En utilisant la représentation par liste de successeurs, écrivez une fonction efficace qui inverse tous les arcs d'un graphe. Quelle est sa complexité ? Déduisez-en une technique pour construire la liste des antécédents de `v` en temps linéaire.

Exercice 5 : Arbre couvrant minimum d'un graphe non-orienté pondéré (25 points)

Un arbre couvrant d'un graphe non-orienté G est un sous-graphe sans cycle de G ayant la même fermeture transitive que G : si un chemin existe entre u et v, il en existe encore un dans l'arbre couvrant. Dans le cas d'un graphe connexe, cela signifie que tous les sommets sont atteints par l'arbre.

On s'intéresse ici à des graphe pondérés où un poids est donné à chaque arête du graphe et on cherche à construire un arbre couvrant minimum : un arbre dont la somme des poids des arêtes est minimale.

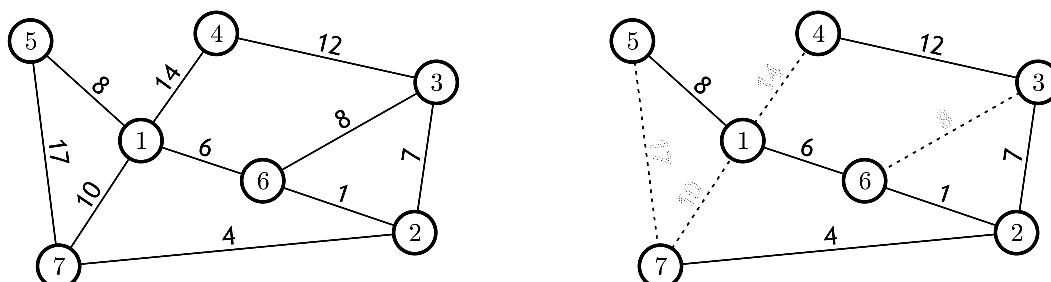


Fig. 1 – À gauche un graphe et à droite son arbre couvrant minimum.

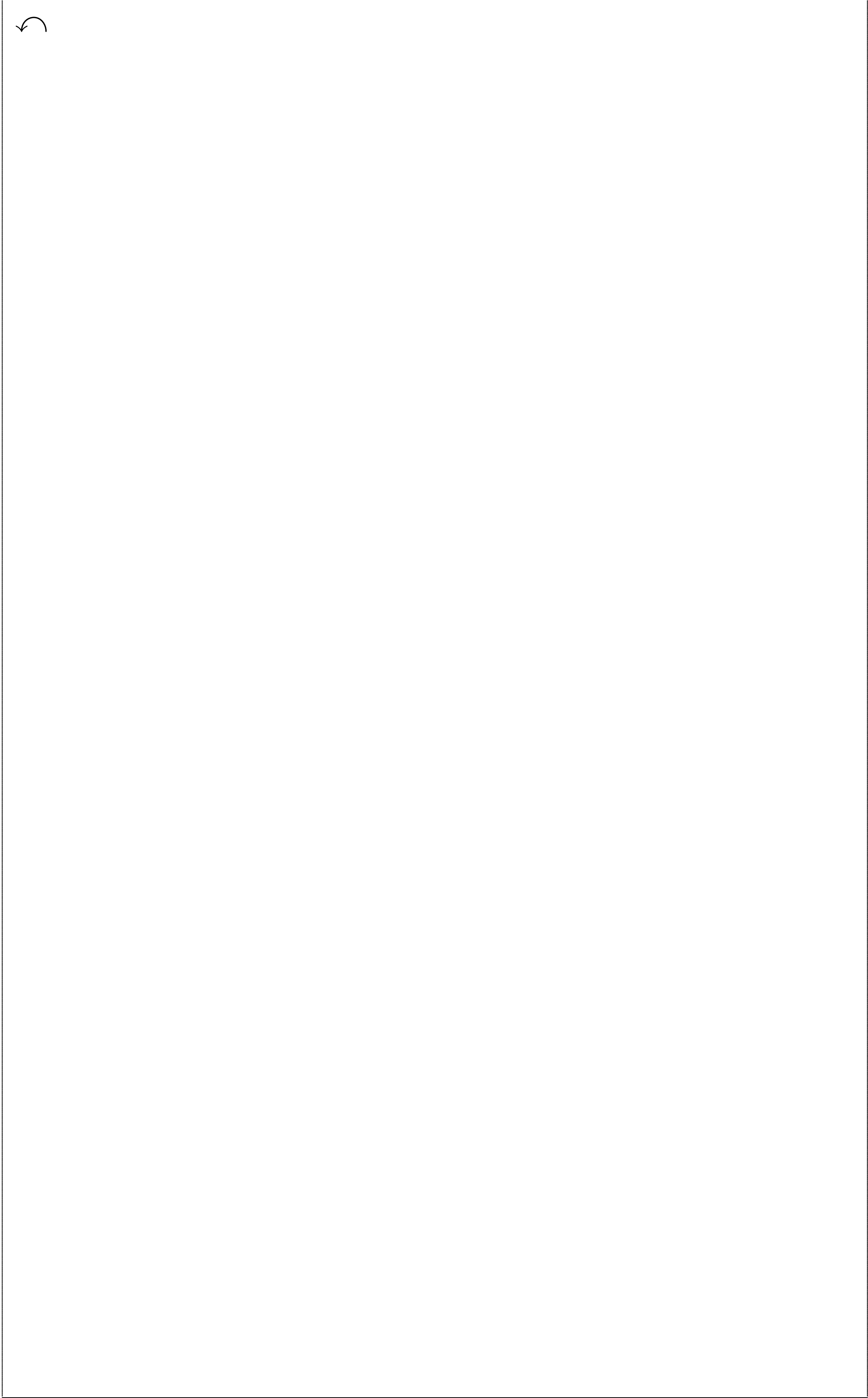
Nous nous intéressons ici à l'algorithme de Prim qui fonctionne de la façon suivante :

- on choisit arbitrairement un sommet qui sera la racine de l'arbre couvrant
- parmi tous les voisins de ce sommet on regarde celui que l'on peut atteindre avec l'arête de plus petit poids
- on ajoute ce sommet à l'arbre avec l'arête qui y mène
- on regarde maintenant l'ensemble des voisins de l'arbre (les voisins des deux sommets) et encore une fois on sélectionne celui que l'on peut atteindre par l'arête de plus petit poids et on l'ajoute à l'arbre
- on recommence ainsi jusqu'à ce que tous les sommets aient été ajoutés à l'arbre.

Afin de pouvoir efficacement sélectionner les prochains sommets à ajouter, on place tous les sommets dans une file de priorité (un tas par exemple), on sélectionne le sommet de priorité la plus élevée et on met à jour les priorités des autres sommets. L'algorithme s'arrête quand tous les sommets ont été sortis de la file.

(25 pts) **5.a]** Écrivez (en pseudo-code) une fonction qui calcule un arbre couvrant minimum (par exemple un tableau des pères des sommets) d'un graphe par l'algorithme de Prim. Intéressez vous en particulier à comment initialiser/mettre à jour les priorités des sommets dans la file de priorité. Quelle est la complexité de cet algorithme (donnez quelques justifications) ?

On considère que l'on peut accéder en temps constant à tous les éléments du graphe : une arête entre deux sommets, la liste des voisins d'un sommet, la liste des arêtes d'un sommet...



Exercice 6 : Intersection de segments (28 points)

On s'intéresse au problème suivant : étant donné un ensemble de n segments dans le plan (donnés par les coordonnées (x_1, y_1) et (x_2, y_2) de leurs extrémités, avec $x_1 < x_2$), on veut savoir si tous les segments sont disjoints ou si deux d'entre eux s'intersectent. On considère que tous les x_1 et x_2 sont distincts : aucun segment n'est vertical et deux segment n'ont jamais deux extrémités ayant la même abscisse.

(5 pts) **6.a**] Écrivez un algorithme qui teste si deux segments donnés s et s' s'intersectent.

Pour tester l'intersection d'un plus grand nombre de segment on va procéder par balayage : une droite verticale parcours le plan de gauche à droite et chaque fois qu'une extrémité de segment est rencontrée on vérifie que des segments "voisins" ne s'intersectent pas (voir Fig. 2). Afin d'obtenir un algorithme efficace on veut maintenir une structure de donnée ordonnée contenant tous les segments qui coupent la droite de balayage. À chaque début de segment on ajoute le segment à cette structure et à chaque fin de segment on l'enlève.

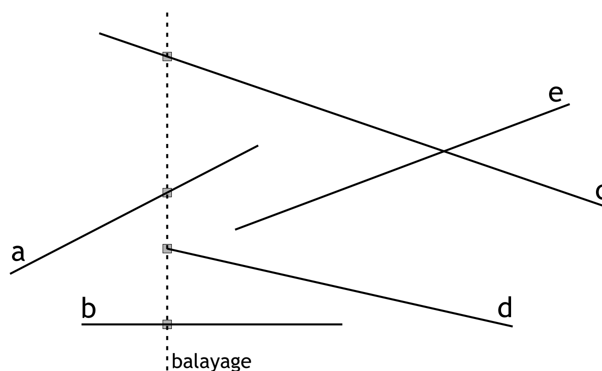


Fig. 2 – Une droite verticale balaye le plan. Au début du segment d, les segments sont dans l'ordre b, d, a, c. Les segments voisins sont donc b et d, d et a, a et c.

(3 pts) **6.b]** Quelle structure de données utiliseriez vous pour avoir une bonne complexité *dans le pire cas* pour les insertions/suppressions ?

(20 pts) **6.c]** Pour ne pas retester l'intersection de mêmes voisins trop souvent, on ne teste que les nouveaux voisins, créés par une insertion ou une suppression. Écrivez un algorithme qui prend en argument un ensemble de segments (sans ordre particulier) et retourne vrai si deux segments s'intersectent et faux sinon. Quelle est sa complexité ?

On ne demande pas de reprogrammer la structure de donnée.

 Ne pas retourner.

Il est interdit de regarder le contenu de ce document avant d'y avoir été invité.

Toute violation de cette interdiction sera sévèrement punie.

 Ne pas retourner.