Conservatoire National des Arts et Métiers
École SITI - Département Informatique

Année 2014–2015

UE NFP 136 (Responsable : Cédric Bentz)
Tous documents autorisés

Examen de l'UE NFP 136 (VARI 2)

22 juin 2015

Exercice 1 : tris et tas

Dans cet exercice, on s'intéresse au tri de tableaux particuliers (tas ou tableaux déjà triés selon un autre ordre).

Travail demandé

1. Écrire une méthode JAVA permettant de trier par ordre décroissant un tableau de n entiers déjà triés par ordre croissant. Donner également la complexité en temps au pire cas de cette méthode (en fonction de n), en cherchant évidemment à la rendre la plus petite possible.

   Le prototype de cette méthode sera le suivant : `public static void triDecroissantTableauCroissant(int[] tab)`, où `tab` est le tableau à trier, et qui sera modifié par la méthode. (On supposera que `tab` est déjà trié par ordre croissant, sans avoir à le vérifier.)

2. Écrire une méthode JAVA permettant de trier par ordre croissant les n entiers stockés dans un tas min, en cherchant à garantir la meilleure complexité en temps possible. Pour cela, on pourra utiliser toute méthode de la classe JAVA `Tas` (vue en cours) qui paraîtra nécessaire (`minimum`, `supprimerMin` ou `inserer`).

   Le prototype de cette méthode sera le suivant : `public static int[] triTasMin(Tas tas)`, où `tas` est le tas min à trier, et le tableau retourné contiendra les n entiers de `tas` triés par ordre croissant.

   Préciser également la complexité en temps au pire cas de cette méthode (en fonction de n).

3. À l'aide des 2 questions précédentes, écrire une méthode JAVA permettant de trier par ordre décroissant les n entiers stockés dans un tas min, en cherchant à garantir la meilleure complexité en temps possible.

   On précisera également (en fonction de n) la complexité en temps au pire cas de cette méthode, dont le prototype sera le suivant : `public static int[] triDecroissantTasMin(Tas tas)`, où `tas` est le tas min à trier, et le tableau d'entiers retourné contiendra les n entiers de `tas` triés par ordre décroissant.

4. Détailler toutes les étapes de l'exécution de la méthode de la question précédente sur le tas à $n = 8$ éléments donné par le tableau suivant :

3	7	4	10	15	17	8	12

Exercice 2 : implémentation du théorème d'Euler

Dans cet exercice, on se propose d'implémenter, pas-à-pas, toutes les étapes permettant de tester si un graphe connexe non orienté possède (ou non) une chaîne eulérienne, à l'aide du théorème d'Euler.

Pour rappel, ce théorème, dû au mathématicien suisse Leonhard Euler, stipule que : *un graphe connexe non orienté possède une chaîne eulérienne si et seulement s'il possède exactement 0 ou 2 sommets de degré impair.*

On rappelle également qu'un graphe à n sommets peut être représenté par une matrice d'adjacence `matAdj` ayant n lignes et n colonnes, et telle que, pour tout couple de sommets i et j, l'élément `matAdj[`$i-1$`][`$j-1$`]` vaut 1 s'il existe un arc entre les sommets i et j (ou bien si les sommets i et j sont voisins, dans le cas d'un graphe non orienté), et 0 sinon.

Par ailleurs, un graphe à n sommets peut aussi être représenté par un tableau de listes d'adjacence `tabListeAdj`, c'est-à-dire un tableau de taille n, dont chaque élément est une liste chaînée (éventuellement vide). Ainsi, pour tout sommet i, l'élément `tabListeAdj[`$i-1$`]` est la liste chaînée des successeurs (ou des voisins, dans le cas d'un graphe non orienté) de i. Par exemple, si le sommet 3 a pour voisins les sommets 1 et 4 dans un graphe non orienté, alors `tabListeAdj[2]` sera représenté par la liste suivante :

```
-> 1 -> 4
```

Travail demandé

1. Implémenter en JAVA une méthode permettant de tester si un graphe donné (représenté par sa matrice d'adjacence) est non orienté ou non. Dans ce cas, sa matrice d'adjacence `matAdj` doit être symétrique, c'est-à-dire que, pour tout i et tout j, l'élément sur la $i$ème ligne et $j$ème colonne doit être égal à l'élément sur la $j$ème ligne et $i$ème colonne.

   Le prototype de la méthode sera le suivant : `public static boolean estNonOriente(int[][] matAdj)`, où le paramètre `matAdj` est la matrice d'adjacence du graphe considéré, et le booléen retourné vaudra *true* si le graphe est non orienté, et *false* sinon.

2. Implémenter en JAVA une méthode permettant de calculer les degrés des sommets d'un graphe non orienté donné sous la forme d'une matrice d'adjacence. Le prototype de cette méthode sera ainsi : `public static int[] degres(int[][] matAdj)`, où `matAdj` est la matrice d'adjacence du graphe considéré, et le $i$ème élément du tableau d'entiers retourné sera égal au degré du $(i+1)$ème sommet.

3. Même question que la précédente, dans le cas d'un graphe non orienté donné sous la forme d'un tableau de listes d'adjacence. Le prototype de cette méthode sera alors le suivant : `public static int[] degres(Liste[] tabListeAdj)`. Détailler également la complexité en temps au pire cas de cette méthode.

4. À l'aide des deux premières questions précédentes, écrire une méthode JAVA permettant de tester, à l'aide du théorème d'Euler, si un graphe non orienté connexe, représenté par sa matrice d'adjacence, admet une chaîne eulérienne ou non.

   On commencera par vérifier que le graphe est bien non orienté (si ce n'est pas le cas, on retournera *false*), puis qu'il est bien connexe (si ce n'est pas le cas, on retournera *false*). Pour tester sa connexité, on supposera l'existence de la méthode suivante (et qui n'est donc pas à implémenter) : `public static boolean estConnexe(int[][] matAdj)`, où `matAdj` est la matrice d'adjacence du graphe considéré, et le booléen retourné vaut *true* si le graphe est connexe, et *false* sinon.

   On calculera ensuite les degrés de ses sommets, puis on vérifiera que le nombre de sommets de degré impair est bien 0 ou 2 (si ce n'est pas le cas, on retournera *false*), et si bien c'est le cas on retournera *true*.

   Enfin, on détaillera la complexité en temps au pire cas de la méthode implémentée, dont le prototype est : `public static boolean admetChaineEulerienne(int[][] matAdj)`, où `matAdj` est la matrice d'adjacence du graphe, et le booléen retourné vaut *true* si le graphe est non orienté, connexe, et possède une chaîne eulérienne, et *false* sinon.

Exercice 3 : édition de liens dans un programme

On considère l'ensemble des modules d'un programme, définis ainsi :

```
module  TRUC        taille  2842
        liens utilisables   MACHIN          620
                            EXIT_PROG       1630
        liens à satisfaire  GET
                            BIDULE
                            SCHMILBLICK
                            ERREUR
        adresse lancement   2546
module  CHAINE      taille  1048
        liens utilisables   SOUS_CHAINE     257
                            DEBUT           657
                            CONCATENER      932
module  GET         taille  760
        liens utilisables   GET             454
module  SHADOK      taille  2320
        liens utilisables   SCHMILBLICK     1215
        liens à satisfaire  DEBUT
                            ERREUR_FATALE
module  UTIL        taille  1132
        liens utilisables   BIDULE          213
                            ERREUR          513
        liens à satisfaire  EXIT_PROG
                            CONCATENER
```

On effectue l'édition de liens de tous ces modules. Répondre aux 5 questions suivantes, en les justifiant brièvement :

1. Donner les adresses d'implantation de ces modules.
2. Donner la taille totale du programme résultant (en octets).
3. Donner l'adresse de lancement du programme résultant.
4. Donner la table des liens.
5. L'édition des liens est-elle correcte ? Justifier.