Conservatoire National des Arts et Métiers
Département Informatique

Année 2017–2018

UE NFP 136 (VARI 2)

Sujet du projet 2017-2018 : Des tas de messages codés

Présentation du projet

L'agent 007 vous le confirmerait : savoir garder ses secrets est le b.a.-ba d'un bon agent secret. Et, en cryptographie, l'une des plus anciennes méthodes de chiffrement de messages est le *code de César* (d'où son nom).

La version chiffrée d'un mot s'obtient en décalant toutes ses lettres d'un décalage d donné. Par exemple, si $d = 3$, le mot "assez" devient "dvvhc" ; autrement dit, 'a' devient 'd', 's' devient 'v', 'e' devient 'h', et 'z' devient 'c'.

Dans ce projet, on se propose de rendre le (dé)chiffrement des mots un peu plus complexe en calculant le décalage indirectement, c'est-à-dire en utilisant une information difficile à deviner a priori, à savoir l'ordre des éléments d'un tas obtenu à l'issue d'un certain nombre d'insertions et de suppressions.

Le but de ce projet est donc d'implémenter en JAVA un algorithme de déchiffrement de messages, ces derniers étant chiffrés à l'aide d'un code de César dont la valeur du décalage est calculée à partir d'un tas. Par ailleurs, les messages à déchiffrer seront fournis sous la forme de fichiers texte.

Ce document a pour objectif de vous guider pas à pas dans les différentes étapes de la réalisation de ce projet, et est à lire attentivement.

Étape 1 : la méthode `lireMessages`

La première ligne de chaque fichier de données (fichier texte) contient 4 informations (données dans cet ordre, et séparées par des virgules) : l, i, s, n. Ici, l désigne le nombre de lignes qui composent le message chiffré contenu dans le fichier (c'est-à-dire le nombre de lignes du fichier, si on exclut la première ligne), i le nombre d'entiers à insérer dans un tas vide (dans l'ordre $1, 2, \ldots, i$), $s \leq i$ le nombre d'éléments à supprimer du tas, et $n \leq i - s$ le numéro du nœud du tas qui contient la valeur du décalage à appliquer.

Chacune des lignes suivantes correspond donc à un bout de message à déchiffrer, et doit pour l'instant être stockée dans une variable de type `String`. La lecture de ce fichier sera réalisée par une méthode nommée

`lireMessages` dans votre classe principale : étant donné qu'il peut y avoir plusieurs lignes dans un fichier, et qu'on a besoin de prendre en compte l'ensemble de ces lignes, cette méthode renverra un tableau d'objets `String` (un pour chaque ligne). Par exemple, considérons le fichier texte suivant :

```
2,4,1,2
Dahhk
Sknhz
```

Ce fichier "msg.txt" possède $l = 2$ lignes (sans compter la première), et le décalage est obtenu en insérant les éléments $1, 2, 3, i = 4$ (dans cet ordre) dans un tas vide, puis en supprimant $s = 1$ élément (le tas obtenu est alors $|2|4|3|$, si on le note sous forme de tableau), et enfin en accédant à la valeur du nœud numéro $n = 2$ (qui est donc 4). À la fin, la méthode renvoie le tableau |"Dahhk"|"Sknhz"| (ce qui correspond à |"Hello"|"World"|, car $d = 4$).

Pour lire des données dans un fichier, on peut utiliser une variable de la classe `BufferedReader`. Voici comment initialiser une variable `lecteur` de cette classe, pour accéder au contenu du fichier "msg.txt" précédent :

```
BufferedReader lecteur = new BufferedReader(new FileReader("msg.txt"));
```

En appelant la méthode `readLine` (qui renvoie une valeur de type `String`) sur cette variable, on lira le contenu de la ligne suivante du fichier de nom "msg.txt" (la lecture d'un fichier étant séquentielle). Ainsi, dans le cas de ce fichier "msg.txt", après le premier appel à cette méthode (qui lit la première ligne du fichier), la variable `ligne` contiendra la valeur `"2,4,1,2"`. Après un deuxième appel à cette méthode (qui lit la deuxième ligne du fichier), la variable `ligne` contiendra la valeur `"Dahhk"` :

```
String ligne = lecteur.readLine(); //ligne contient "2,4,1,2"
ligne = lecteur.readLine(); //ligne contient "Dahhk"
```

La construction d'une variable de type `BufferedReader` est susceptible de générer une exception (si le fichier de nom "msg.txt" n'existe pas, par exemple), et il serait donc préférable de gérer toutes ces instructions dans un bloc `try{...} catch(Exception e){...}`. Enfin, après utilisation, cette variable `lecteur` doit être fermée, à l'aide de l'instruction `lecteur.close();`.

Pour lire le fichier "msg.txt", il est également possible d'utiliser la classe `Scanner`, de la façon suivante :

```
Scanner lecteur = new Scanner(new File("msg.txt"));
```

On peut alors lire chaque ligne à l'aide de la méthode `nextLine()` :

```
String ligne = lecteur.nextLine(); //premier appel : ligne="2,4,1,2"
ligne = lecteur.nextLine(); //deuxième appel : ligne="Dahhk"
```

Pour stocker les informations lues sur chaque ligne dans les variables correspondantes, on vous rappelle l'existence de :

- la méthode `split()` de la classe `String`, qui "découpe" la variable de type `String` sur laquelle la méthode est appelée en morceaux séparés par le symbole passé en paramètre (par exemple ","), et stocke les morceaux obtenus dans le tableau d'éléments de type `String` renvoyé,
- la méthode de classe `parseInt()` de la classe `Integer`, qui permet de convertir en entier (variable de type `int`) l'objet de la classe `String` passé en paramètre (par exemple, "2" sera converti en l'entier 2).

(Pour plus d'informations sur ces méthodes, se référer à l'énoncé du TP 1.)

Étape 2 : la méthode `codageBinaire`

On se propose à présent d'écrire une méthode `codageBinaire` (qui sera utile pour rechercher un nœud dans un tas, et qui pourra par exemple être intégrée à votre classe principale), qui prend en paramètre un entier n et renvoie un tableau d'entiers contenant les bits résultant de l'écriture binaire de n. Par exemple, si $n = 13 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$, le tableau renvoyé par la méthode, de taille 4 (puisqu'il y a 4 bits), sera le suivant : $|1|1|0|1|$.

Il faut commencer par calculer (à l'aide d'une boucle) la taille du tableau (c'est-à-dire le nombre de bits nécessaires pour écrire n), qui correspond à l'exposant de la plus petite puissance de 2 qui excède (strictement) la valeur de n (ainsi, $2^4 > n = 13 \geq 2^3$). On divise ensuite n par 2 (division *euclidienne*, autrement dit *entière*), et on garde en mémoire le reste (0 ou 1), obtenu grâce à l'opérateur modulo (%). On redivise le quotient de cette division par 2 (de nouveau, on garde en mémoire le reste), et on recommence le processus tant que le quotient n'est pas nul. La suite des restes obtenus fournit les bits résultant de l'écriture binaire de n, mais **dans l'ordre inverse**. Par exemple, pour $n = 13$, on obtiendra les restes 1, 0, 1, puis 1.

Étape 3 : la classe `TasParChainage`

On se propose dans cette étape d'écrire une classe `TasParChainage`, qui implémente les opérations d'un tas min (accéder à l'élément de valeur minimum, insérer un élément, et supprimer l'élément de valeur minimum) représenté sous la forme de chainages (et non à l'aide d'un tableau). On aura également besoin d'une méthode supplémentaire, qui permettra d'accéder à un nœud du tas (ou à son père) en connaissant son numéro.

Chaque objet (variable) de cette classe représentera un nœud du tas considéré (le tas complet étant donné par la référence vers sa racine), et contiendra 5 variables d'instance (appelées aussi *attributs*) ; on rappelle que le bon usage en JAVA est de les déclarer comme `private` :

- Trois références : une vers le père du nœud courant, une vers son fils gauche, et une vers son fils droit,
- Un entier représentant la donnée contenue dans le nœud,
- Un entier dont la valeur est égale au nombre de sommets du tas qui se trouvent sous le nœud courant ; cette information ne sera utile qu'à la racine du tas, puisque dans ce cas cet entier sera égal à la taille du tas.

On implémentera d'abord une méthode qui permet d'accéder à un nœud du tas (ou à son père, sauf si le nœud considéré est la racine), à partir de son numéro *num* (les sommets étant supposés numérotés de la racine vers les feuilles, et de gauche à droite). Pour cela, on se placera à la racine et on commencera par déterminer l'écriture binaire de *num* (à l'aide de la méthode `codageBinaire`, implémentée à l'étape précédente), puis on se déplacera (en considérant les bits par valeurs décroissantes des puissances de 2 associées, sans tenir compte de celui associé à la plus grande puissance de 2) : vers le fils gauche si le bit considéré vaut 0, vers le fils droit sinon. (Si on veut accéder au père du nœud et non au nœud lui-même, ce qui dépendra de la valeur d'un booléen passé en paramètre, il suffit de s'arrêter un cran avant.)

On implémentera ensuite les méthodes de base sur les tas min : accéder à l'élément de valeur minimum (c'est-à-dire à la valeur qu'on trouve à la racine du tas), insérer un élément, et supprimer l'élément de valeur minimum.

Pour insérer un élément, on commence par incrémenter la taille du tas, et on détermine ensuite (à l'aide de la méthode décrite précédemment) le père de la dernière feuille (celle qui va contenir la valeur insérée). Si ce père ne possède pas déjà un fils gauche, alors la feuille contenant la valeur insérée sera son fils gauche, sinon cette feuille sera son fils droit. Puis, tant que la valeur insérée n'est pas à sa place dans le tas (c'est-à-dire est strictement plus petite que la valeur de son père), alors il faut la faire remonter (c'est-à-dire échanger cette valeur avec celle de son père), comme vu en cours.

Pour supprimer l'élément de valeur minimum, on détermine (à l'aide de la méthode décrite avant l'insertion) le père de la dernière feuille (on peut alors décrémenter la taille du tas) : si ce père ne possède pas de fils droit, alors la dernière feuille est son fils gauche ; sinon, il s'agit de son fils droit. On remplace alors la valeur à la racine du tas par celle de la dernière feuille, on supprime cette dernière feuille, puis on cherche la place dans le tas de la valeur remontée à la racine. Pour ce faire, on la fait descendre niveau par niveau dans le tas, en l'échangeant avec la valeur du plus petit fils (si cette dernière est strictement plus petite) tant que nécessaire, comme vu en cours.

À l'issue de cette étape (qui peut être réalisée et testée de façon totalement indépendante du reste du projet), on a alors une classe permettant de manipuler des tas obtenus par chainages, et on peut également accéder facilement à la valeur du $j$ème nœud d'un tel tas. On peut donc à présent récupérer les messages à déchiffrer, et calculer le décalage grâce à ces tas.

Étape 4 : la méthode `decoder` (ou `dechiffrer`)

Cette dernière étape consiste à écrire une méthode qui prend en paramètre un message à déchiffrer (sous la forme d'une variable de type `String`) et un décalage (valeur entière), et qui renvoie le message déchiffré grâce au décalage (également sous la forme d'une valeur de type `String`).

Pour ce faire, on parcourra le message à déchiffrer lettre par lettre, et pour chacune on lui appliquera le décalage donné. Par exemple, si la lettre lue est `'f'`, et si on veut appliquer un décalage de 2, on peut écrire :

```
char c = 'f'+2;
```

Le caractère `c` obtenu aura alors la valeur `'h'`. En réalité, il faut aussi prendre en compte le cas où on est obligé de "faire le tour" : ainsi, `'w'` décalé de 5 doit donner `'b'`. Pour ce faire, on pourra réutiliser l'opérateur modulo (%), et le fait que toutes les lettres minuscules non accentuées sont consécutives (ainsi, tester si un caractère `c` est une lettre minuscule non accentuée revient à vérifier si `c` $\geq$ `'a'` et `c` $\leq$ `'z'`). On peut donc écrire :

```
char c = 'a'+('w'-'a'+5)%26;
```

Le caractère `c` obtenu vaudra alors `'b'`. On supposera que les messages à décoder ne peuvent contenir que des lettres minuscules non accentuées, des lettres majuscules non accentuées (qui sont elles aussi toutes consécutives, et comprises entre `'A'` et `'Z'`), ainsi que les caractères `'-'` (tiret) et `' '` (espace), qu'on laissera tels quels. On rappelle en particulier l'existence de la méthode `charAt()` de la classe `String`, qui renvoie le caractère (variable de type `char`) qui se trouve, dans la variable de type `String` sur laquelle la méthode est appelée, à l'indice (entier) passé en paramètre.

Touches finales

Pour compléter le projet, il reste à écrire le (court) programme principal. Ce dernier doit récupérer un paramètre au moment de l'appel du programme : le nom du fichier texte dans lequel est stocké le message à décoder.

Il faut dans un premier temps lire ce fichier, pour récupérer l'ensemble du message à décoder sous la forme d'un tableau de `String` (cf Étape 1).

Dans un second temps, il faut insérer dans un tas vide les éléments $1, \ldots, i$, supprimer ensuite s éléments, puis déterminer la valeur du décalage d, égale à la valeur du nœud numéro n dans le tas obtenu (cf Étape 3).

Enfin, il faut déchiffrer l'ensemble du message à l'aide du décalage d (cf Étape 4), et afficher le message ainsi décodé.

Ainsi, voici ce qu'afficherait le programme si on lui fournit en paramètre, au moment de l'appel, le nom du fichier donné en exemple en page 2 :

```
> java ClassePrincipaleProjet msg.txt

Message codé :

Dahhk
Sknhz

==> Message décodé (valeur du décalage d=4) :

Hello
World
```

Le projet est à rendre par e-mail, à l'adresse : `cedric.bentz@cnam.fr`. Lors du rendu, il faudra bien évidemment fournir le code source commenté de toutes vos classes (fichiers .`java`), et leur bytecode (fichiers .`class`).