Conditions de distribution et de copie

Exercices (III)

Fabrice Rossi

19 décembre 2000

1 Chaînes de caractères

1.1 Analyse de programmes

Exercice 1.1 :

On considère la méthode suivante :

```
1 public static String e(String a) {
2   String r="";
3   for(int i=0;i<a.length();i++)
4     r=a.charAt(a.length()-i)+r;
5   return r;
6 }
```

1. Cette méthode :
   (a) renvoie le miroir de la chaîne paramètre (c'est-à-dire la chaîne constituée des mêmes caractères que la chaîne paramètre mais dans le sens inverse)
   (b) renvoie une chaîne identique à la chaîne paramètre
   (c) plante quand la chaîne paramètre contient au moins un caractère
   (d) plante toujours
2. On remplace la ligne 3 par la ligne `for(int i=1 ;i<=a.length() ;i++)`. La nouvelle méthode :
   (a) renvoie le miroir de la chaîne paramètre (c'est-à-dire la chaîne constituée des mêmes caractères que la chaîne paramètre mais dans le sens inverse)
   (b) renvoie une chaîne identique à la chaîne paramètre
   (c) plante quand la chaîne paramètre contient au moins un caractère
   (d) plante toujours
3. On remplace la ligne 3 par la ligne `for(int i=a.length() ;i>0 ;i--)`. La nouvelle méthode :
   (a) renvoie le miroir de la chaîne paramètre (c'est-à-dire la chaîne constituée des mêmes caractères que la chaîne paramètre mais dans le sens inverse)
   (b) renvoie une chaîne identique à la chaîne paramètre
   (c) plante quand la chaîne paramètre contient au moins un caractère
   (d) plante toujours

4. Dans la méthode **d'origine**, on remplace la ligne 4 par :
   `r=a.charAt(a.length()-i-1)+r ;`.
   La nouvelle méthode :
   (a) renvoie le miroir de la chaîne paramètre (c'est-à-dire la chaîne constituée des mêmes caractères que la chaîne paramètre mais dans le sens inverse)
   (b) renvoie une chaîne identique à la chaîne paramètre
   (c) plante quand la chaîne paramètre contient au moins un caractère
   (d) plante toujours

Exercice 1.2 :

On considère la méthode suivante :

```
1  public static int d(String a,String b) {
2    int k=Math.min(a.length(),b.length());
3    for(int i=0;i<k;i++) {
4      if(a.charAt(i)<b.charAt(i)) {
5        return -1;
6      } else {
7        if (a.charAt(i)>b.charAt(i)) {
8          return 1;
9        }
10     }
11   }
12   if(a.length()<b.length()) {
13     return -1;
14   } else {
15     if(a.length()>b.length()) {
16       return 1;
17     } else {
18       return 0;
19     }
20   }
21 }
```

1. Cette méthode renvoie :
   (a) -1 si la chaîne `a` est **avant** la chaîne `b` dans le dictionnaire, 1 si `a` est **après** `b` et 0 si les chaînes sont identiques
   (b) -1 si la chaîne `a` est **après** la chaîne `b` dans le dictionnaire, 1 si `a` est **avant** `b` et 0 si les chaînes sont identiques
   (c) -1 si le premier caractère de `a` est **avant** le premier caractère de `b` dans l'alphabet, 1 si le premier caractère de `a` est **après** le premier caractère de `b` et 0 si les deux caractères sont égaux
   (d) -1 si le premier caractère de `a` est **après** le premier caractère de `b` dans l'alphabet, 1 si le premier caractère de `a` est **avant** le premier caractère de `b` et 0 si les deux caractères sont égaux

2. On remplace la ligne 2 de la méthode par `int k=Math.max(a.length(),b.length()) ;`. Suite à cette modification, la méthode fonctionne *en général* exactement comme la méthode d'origine. Il arrive cependant que la méthode plante. Donner un exemple de chaînes paramètres qui provoquent un plantage. Donner une caractérisation précise de tous les cas qui entraînent un plantage.
3. Dans la méthode **d'origine**, on remplace les lignes 12 à 20 par la ligne `return a.length()-b.length() ;` :
   (a) la méthode ne compile plus
   (b) la méthode peut parfois planter (provoquer l'arrêt du programme)
   (c) la méthode fonctionne comme la version d'origine
   (d) le signe du résultat de la nouvelle méthode est identique au signe du résultat de la méthode d'origine
   (e) la méthode fonctionne mais donne des résultats complètement différents de ceux de la méthode d'origine

Exercice 1.3 :

On considère la méthode :

```
1   public static String f(String s) {
2     StringBuffer sb=new StringBuffer(s);
3     int i;
4     for(i=0;i<sb.length();i++) {
5       char c=sb.charAt(i);
6       sb.setCharAt(i,sb.charAt(sb.length()-i-1));
7       sb.setCharAt(sb.length()-i-1,c);
8     }
9     return sb.toString();
10  }
```

1. La méthode devrait renvoyer le miroir de la chaîne paramètre, c'est-à-dire une chaîne de caractères obtenue en retournant la chaîne paramètre. En fait, la méthode ne fonctionne pas :
   (a) elle renvoie une chaîne égale à la chaîne d'origine
   (b) elle ne compile pas
   (c) elle fonctionne seulement avec les chaînes de longueur strictement supérieure à 1 et plante sinon
   (d) elle plante toujours
2. L'erreur principale est située à la ligne :
   (a) 4
   (b) 6
   (c) 5
   (d) 7
3. Donner une version correcte de cette méthode, en modifiant le moins possible la version proposée. Il est interdit d'utiliser la méthode `reverse` des `StringBuffer`s.

Exercice 1.4 :

Indiquer l'affichage produit par le programme suivant :

Analyse

```
1  public class Analyse {
2    public static void modif1(int x) {
3      x+=2;
4    }
5    public static void modif2(int x) {
6      x=5;
7    }
8    //
9    public static String ajoute1(String s) {
10     s+=" toto";
11     return s;
12   }
13   public static String ajoute2(String s) {
14     s="toto";
15     return "titi";
16   }
17   //
18   public static StringBuffer concat1(StringBuffer sb) {
19     sb.append(" +++");
20     return sb;
21   }
22   public static StringBuffer concat2(StringBuffer sb) {
23     sb= new StringBuffer("+++");
24     return sb;
25   }
26   public static void main(String[] args) {
27     int x=24;
28     modif1(x);
29     System.out.println(x);
30     x=24;
31     modif2(x);
32     System.out.println(x);
33     //
34     String s="titi";
35     System.out.println(ajoute1(s));
36     System.out.println(s);
37     s="titi";
38     System.out.println(ajoute2(s));
39     System.out.println(s);
40     //
41     StringBuffer sb=new StringBuffer("---");
42     System.out.println(concat1(sb));
43     System.out.println(sb);
44     sb=new StringBuffer("---");
45     System.out.println(concat2(sb));
46     System.out.println(sb);
```

```
47    }
48  }
```

1.2 Programmation

Exercice 1.5 :

Ecrire une méthode qui compare deux chaînes de caractères au regard de l'ordre lexicographique (l'ordre du dictionnaire pour les caractères non accentués). On n'utilisera pas la méthode `compareTo` mais on cherchera à produire des résultats identiques à ceux de cette méthode.

Exercice 1.6 :

Ecrire une méthode qui à une chaîne de caractères associe le nombre de caractères distincts que celle-ci contient.

Exercice 1.7 :

Ecrire une méthode qui permet la saisie d'une chaîne de caractères ne contenant que des chiffres. Ecrire une méthode qui à une chaîne de caractères ne contenant que des chiffres associe le nombre correspondant (sous forme d'un `int`). Ecrire une méthode réalisant l'opération inverse.

Exercice 1.8 :

Écrire un programme qui demande à l'utilisateur un texte et un mot, puis affiche pour chaque lettre du mot, le nombre d'occurrences de cette lettre dans le texte de départ.

Exercice 1.9 :

Ecrire une méthode qui à une chaîne de caractères associe le nombre de mots que celle-ci contient. Ecrire ensuite une méthode qui calcule la longueur moyenne des mots d'un texte.

Exercice 1.10 :

Ecrire une méthode qui à une chaîne de caractères associe `true` si et seulement si la phrase qu'elle contient vérifie les règles élémentaires de typographie : la phrase commence par une majuscule et termine par un point. Les mots sont séparés par exactement un espace (sauf en cas de symbole de ponctuation). La virgule et le point sont collés au mot qui les précède et sont suivis par un espace. Le point-virgule et les deux points sont précédés et suivis par un espace.

Exercice 1.11 :

Ecrire une méthode qui reçoit une chaîne de caractères de longueur impaire et l'affiche sous la forme d'un sablier et d'un noeud papillon. Par exemple :

```
bonjour          b     r
 onjou           bo   ur
  njo            bon our
   j             bonjour
  njo            bon our
 onjou           bo   ur
bonjour          b     r
```

Exercice 1.12 :

Ecrire une méthode qui à deux chaînes de caractères `s1` et `s2` associe la chaîne constituée des caractères de `s1` qui n'apparaissent pas dans `s2` (dans le même ordre).

Exercice 1.13 :

Ecrire une méthode qui reçoit en paramètre une chaîne de caractères et l'affiche sur deux lignes, en affichant sur la première ligne les caractères de rang impair et sur la seconde les caractères de rang pair. Par exemple, pour `"ALGORITHME"`, on obtient :

```
A G R T M
 L O I H E
```

Exercice 1.14 :

On donne l'algorithme de cryptage élémentaire (et très inefficace) suivant :

Données :
- une chaîne de caractères `t` : le texte à coder ;
- une chaîne de caractères `c` : la clé (de longueur strictement inférieure à celle de `t`).

Résultat : une chaîne de caractères cryptée.

1. créer une chaîne vide `s`.
2. pour chaque caractère `x` de `t` (dans l'ordre) :
   (a) extraire de `c` le caractère de position `i` définie comme suit : soit `j` la position de `x` dans `t`. `i` est le reste de la division de `j` par la longueur de `c`.
   (b) ajouter au code unicode de `x` celui de `c`.
   (c) ajouter à la chaîne `s` le caractère correspondant au code unicode qui vient d'être obtenu.
3. Résultat : la chaîne `s`.

Programmez une méthode de codage et une méthode de décodage utilisant cet algorithme.

Exercice 1.15 :

Etant donnés une chaîne de caractères `s` et un caractère `c`, on peut découper `s` selon `c` de la manière suivante : si `s` ne contient pas `c`, l'unique sous-chaîne u_0 est la chaîne `s` entière. Sinon, `s` est formée d'une première partie u_0 ne contenant pas `c`, suivie de `c`, suivie d'une seconde partie s_1. Si s_1 n'est pas la chaîne vide, on peut recommencer l'opération sur celle-ci, en cherchant `c` dans s_1, ce qui permet de construire u_1 et s_2, etc.

Ecrire une méthode qui à une chaîne de caractères `s`, un caractère `c` et un entier `n` associe la sous-chaîne u_n (éventuellement vide).

1.3 Problèmes

Problème 1.16 :

Pour répondre aux questions de ce problème, on pourra en général donner deux solutions : une avec les `String`s, l'autre avec les `StringBuffer`s. Les deux solutions sont intéressantes à comparer, tant au niveau de l'efficacité que de la simplicité de l'écriture.

1. Écrire une méthode `build` prenant comme paramètres un caractère `c` et un entier positif `n`, et renvoyant la chaîne constituée de `n` fois le caractère `c`. Si le paramètre entier est négatif ou nul, la méthode devra renvoyer la chaîne de caractères vide.

   Exemple : le résultat de `build('a',3)` est donc la chaîne de caractères `"aaa"`.

2. Écrire une méthode `combine` prenant comme paramètres deux caractères `a` et `b`, et deux entiers `n` et `p`. En utilisant la méthode `build`, `combine` devra associer à ces paramètres la chaîne de caractères constituée de `n` fois `a`, suivi de `p` fois `b`, puis encore `n` fois `a`.

   Exemple : le résultat de `combine('u','V',3,2)` est donc la chaîne `"uuuVVuuu"`. La méthode doit fonctionner même si `n` ou `p` est nul.

3. Écrire une méthode `pyramide` prenant comme paramètre un entier strictement positif `n`. Cette méthode devra afficher une pyramide de hauteur `n` à l'écran (en utilisant **impérativement** la méthode `combine`). Un tel affichage comprend `n` lignes. La première ligne comprend `n-1` espaces, un `+` et encore `n-1` espaces. Chaque ligne est obtenue en remplaçant par des `+` les espaces qui entourent la chaîne des `+` dans la ligne précédente. Par exemple, la seconde ligne comporte `n-2` espaces, suivis de 3 `+`, suivis de `n-2` espaces. La dernière `ligne` contient donc `2n-1` fois le caractère `+`.

   Exemple : ceci donne pour `n` égal à 4 l'affichage suivant :

   ```
   ___+___
   __+++__
   _+++++_
   +++++++
   ```

   Pour rendre l'affichage plus clair, les espaces ont été remplacés par le caractère `_` (on utilisera cette notation dans tout l'énoncé).

4. Calculer (en justifiant mathématiquement votre résultat) u_n, le nombre de caractères `+` affiché par l'appel `pyramide(n)`.

5. Le caractère `'\n'` s'affiche à l'écran sous forme d'un passage à la ligne (il s'agit dien d'un seul caractère). Si on exécute par exemple :

   ```
   System.out.println("a\nb") ;
   ```

   On obtiendra l'affichage suivant :

   ```
   a
   b
   ```

   En utilisant cette propriété, écrire une méthode `pyramide2` qui effectue les mêmes opérations que `pyramide`, mais place le dessin de la pyramide dans une chaîne de caractères au lieu de l'afficher, et renvoie la chaîne résultat.

   Exemple : le résultat de `pyramide2(2)` sera donc la chaîne `_+_\n+++` (dans laquelle `_` remplace le caractère espace).

6. Écrire une méthode `compte` qui à une chaîne de caractères `s` et un caractère `c` associe le nombre d'apparitions de `c` dans `s`.

   Exemple : l'appel `compte("bbbAABBb",'b')` renvoie donc 4.

7. Écrire une méthode `remplace` prenant comme paramètres une chaîne de caractères `s`, un entier positif ou nul `n` et deux caractères `a` et `b`. La méthode devra renvoyer une nouvelle chaîne obtenue en remplaçant dans `s` la `n`-ième apparition du caractère `a` par le caractère `b`.

   Exemple : l'appel `remplace("AAABBBAA",2,'B','-')` donne ainsi comme résultat la chaîne `"AAAB-BAA"`.

8. Écrire une méthode `modifie` prenant comme paramètres une chaîne de caractères `s` et deux caractères `a` et `b`. La méthode renvoie une chaîne de caractères obtenue en remplaçant **une** apparition de `a` par `b` dans `s`, choisie aléatoirement. On utilisera impérativement les méthodes `random` (de la classe `Math`) et `remplace`.

   Exemple : le résultat de l'appel `modifie("AAABBBAA",'B','-')` pourra être `"AAA-BBAA"`, `"AAAB-BAA"` ou `"AAABB-AA"`, selon l'apparition choisie.

9. Écrire une méthode `pyramide3` prenant comme paramètre un entier `n`. Cette méthode commencera par utiliser `pyramide2` pour fabriquer une pyramide dans une chaîne de caractères, puis elle appellera plusieurs fois `modifie` afin de remplacer 25% des caractères `+` par `*`. Elle aura pour résultat la chaîne de caractères ainsi obtenue.

   Exemple : comme `modifie` est aléatoire, nous n'indiquons ici qu'un seul résultat possible pour l'appel `pyramide3(3)`, à savoir la chaîne

   ```
   __+__\n_*++_\n++++*+
   ```

   Cette chaîne s'affiche de la façon suivante :

   ```
   __+__
   _*++_
   +++*+
   ```

   Notons bien que `pyramide3` **ne fait pas d'affichage**.

10. Écrire une méthode `développe` prenant comme paramètre deux chaînes de caractères `s` et `t`, et un caractère `c`. Le résultat de la méthode est la chaîne `s` dans laquelle chaque apparition de `c` a été remplacée par la chaîne `t`.

    Exemple : le résultat de `développe("bAbAAd","XoY",'A')` est donc la chaîne `"bXoYbXoYXoYd"`.

11. On appelle p_n la chaîne de caractères produite par l'appel `pyramide2(n)`, c'est-à-dire la pyramide de hauteur `n`. On peut montrer que p_{n+1} s'obtient à partir de p_n par l'algorithme suivant :

    (a) `u` s'obtient en ajoutant un espace au début et à la fin de p_n ;

    (b) `v` est le résultat du remplacement dans `u` de `\n` par la chaîne `"_\n_"` (l'espace est ici remplacé par `"_"`) ;

    (c) p_{n+1} s'obtient enfin en ajoutant à `v` le caractère `\n` puis la dernière ligne de la pyramide (celle qui ne contient que des `+`).

    Programmer une méthode `développe2` qui applique à une chaîne `s` et un entier `n` l'algorithme qui vient d'être présenté, et renvoie la chaîne résultat. Pourquoi le paramètre `n` est-il indispensable ? Notons qu'il correspond à p_n et que l'appel `développe2(`p_n`,n)` doit donc produire comme résultat la chaîne p_{n+1}.

12. Programmer une méthode `pyramide4` qui produit le même résultat que `pyramide2` mais en utilisant la méthode `développe2`.

13. Programmer une méthode `randomChar`, prenant comme paramètre un entier `n` et renvoyant soit le caractère `+`, soit le caractère `*`. Plus précisément, la méthode choisit aléatoirement un entier compris entre 1 et `n` et renvoie `*` si et seulement si cet entier est égal à `1`. En moyenne on obtient donc le caractère `+` `n-1` fois sur `n`. Programmer une méthode `combineAléa` prenant comme paramètres deux entiers `n` et `p` et fabrique une chaîne de `p` caractères obtenue en utilisant `randomChar` avec comme paramètre `n`. Programmer enfin une méthode `pyramide5` qui obtient directement un résultat similaire à `pyramide3`. Quelles sont les différences entre les résultats de deux méthodes ?

Problème 1.17 :

On souhaite étudier dans une chaîne de caractères les séquences de caractères identiques, c'est-à-dire les séquences de la forme `"aaaa"`. Il faut bien sûr qu'une séquence de caractères identiques comporte au moins deux caractères pour qu'elle soit prise en compte !

Questions :

1. Écrire une méthode qui à une chaîne de caractères associe une nouvelle chaîne de caractères obtenue en remplaçant chaque séquence de lettres identiques par la première d'entre elles (par exemple `"aabb"` devient `"ab"`, `"aaabbc"` devient `"abc"`, etc.).
2. Écrire une méthode qui à une chaîne de caractères associe le nombre de séquences de caractères identiques qu'elle contient (pour `"aaabbc"` on obtient 2).
3. Écrire une méthode qui à une chaîne de caractères associe un booléen `true` si elle contient au moins une séquence de caractères identiques et `false` sinon. On **n'utilisera pas** la méthode solution de la question précédente.
4. Écrire une méthode qui à une chaîne de caractères associe la longueur de la plus longue séquence de caractères identiques qu'elle contient et 0 si elle n'en contient pas (pour `"aaabbc"` on obtient 3).
5. Écrire une méthode qui à une chaîne de caractères associe la moyenne des longueurs des séquences de caractères identiques qu'elle contient et 0 si elle n'en contient pas. Pour la chaîne `"aaabbcbbaad"`, on obtient 2.25. En effet, elle contient 4 séquences, trois de 2 lettres et une de 3 lettres, et la moyenne des longueurs est bien 2.25.

Problème 1.18 :

Dans ce problème, on programme un ensemble de méthodes qui permettent d'évaluer une expression contenue dans une chaîne de caractères :

1. écrire une méthode qui à une chaîne de caractères associe `true` si et seulement si elle ne contient que des chiffres, des lettres et des symboles mathématiques (`+`, `/`, `*`, `-` et les parenthèses) ;
2. écrire une méthode qui à une chaîne de caractères associe `true` si elle contient une expression correctement parenthèsée (à chaque parenthèse ouvrante correspond une parenthèse fermante), et `false` sinon ;
3. écrire une méthode qui à une expression correctement parenthèsée associe la chaîne de caractères contenu dans la première paire de parenthèses ne contenant pas d'autre paire de parenthèses. Si on a par exemple la chaîne `((a+(b+c))+(a*b))`, on obtient `b+c` ;
4. écrire une méthode qui calcule la valeur numérique d'une chaîne de caractères contenant un calcul élémentaire de la forme "opérande numérique 1 opérateur opérande numérique 2", où les opérandes numériques sont des nombres entiers ;
5. écrire une méthode qui calcule la valeur numérique d'une chaîne de caractères contenant un calcul comportant éventuellement des parenthèses mais dans lequel chaque expression sans parenthèse se réduit à un calcul de la forme élémentaire utilisé dans la question précédente.

2 Tableaux

2.1 Analyse de programmes

2.1.1 Effets de bord

Exercice 2.1 :

Indiquer l'affichage produit par le programme suivant :

Analyse2

```
1  public class Analyse2 {
2    public static String toString(double[] tab) {
3      StringBuffer sb=new StringBuffer("{");
4      for(int i=0;i<tab.length-1;i++)
5        sb.append(tab[i]).append(',');
6      if(tab.length>0)
7        sb.append(tab[tab.length-1]);
8      sb.append('}');
9      return sb.toString();
10   }
11   public static String toString(double[][] tab) {
12     StringBuffer sb=new StringBuffer("{");
13     for(int i=0;i<tab.length-1;i++)
14       sb.append(toString(tab[i])).append(",\n ");
15     if(tab.length>0)
16       sb.append(toString(tab[tab.length-1]));
17     sb.append('}');
18     return sb.toString();
19   }
20   public static void gnark(double[][] tab) {
21     for(int i=0;i<tab.length;i++)
22       if(i<tab[i].length)
23         tab[i][i]+=1;
24   }
25   public static void blurf(double[][] tab) {
26     double[][] tab2=(double[][])tab.clone();
27     gnark(tab2);
28   }
29   public static void plouf(double[][] tab) {
30     tab[0]=new double[] {1,-2,7};
31   }
32   public static void groumpf(double[][] tab) {
33     double[][] tab2=(double[][])tab.clone();
34     plouf(tab2);
35   }
36   public static void pouic(double[][] tab) {
37     double[][] tab2=(double[][])tab.clone();
38     for(int i=0;i<tab2.length;i++) {
39       tab2[i]=(double[])tab2[i].clone();
40     }
```

```
41      gnark(tab2);
42    }
43    public static void main(String[] args) {
44      double[][] tab={{2,3,4},{-1,4,5}};
45      System.out.println(toString(tab));
46      //
47      gnark(tab);
48      System.out.println(toString(tab));
49      //
50      tab=new double[][] {{1,2,-1},{-1},{1,2,-1}};
51      gnark(tab);
52      System.out.println(toString(tab));
53      //
54      tab=new double[][] {{1,2,-1},{-1},{1,2,-1}};
55      blurf(tab);
56      System.out.println(toString(tab));
57      //
58      tab=new double[][] {{2,-3,4},{-1,4,-5}};
59      plouf(tab);
60      System.out.println(toString(tab));
61      //
62      tab=new double[][] {{-2,3,4},{3,4,6}};
63      groumpf(tab);
64      System.out.println(toString(tab));
65      //
66      tab=new double[][] {{7,4},{-1,3,6}};
67      pouic(tab);
68      System.out.println(toString(tab));
69    }
70  }
```

2.1.2 Méthodes

Exercice 2.2 :

On considère la méthode :

```
1  public static boolean a(int[] x) {
2    for(int i=1;i<x.length;i++) {
3      if(x[i-1]>x[i]) {
4        return false;
5      }
6    }
7    return true;
8  }
```

1. si le tableau paramètre est de longueur strictement supérieure à 1, cette méthode :
   (a) renvoie `true` si et seulement si le tableau `x` est rangé en ordre croissant
   (b) renvoie `true` si et seulement si le tableau `x` est rangé en ordre décroissant

(c) renvoie `true` si et seulement si le tableau `x` est rangé en ordre croissant à partir de sa case numéro 1

(d) renvoie `true` si et seulement si le tableau `x` est rangé en ordre décroissant à partir de sa case numéro 1

2. Si on l'appelle avec comme paramètre un tableau à une seul case, cette méthode :

(a) renvoie `false`

(b) renvoie `true`

(c) plante

Exercice 2.3 :

On considère la méthode :

```
1  public static boolean b(int[] y,int[] x) {
2    if(y.length>x.length) {
3      return false;
4    }
5    int k=0;
6    for(int i=0;i<x.length;i++) {
7      if(x[i]==y[k]) {
8        k++;
9        if(k==y.length) {
10         return true;
11       }
12     }
13   }
14   return false;
15 }
```

1. Cette méthode :

(a) renvoie `true` si et seulement si tous les éléments de `x` apparaissent dans `y` dans le même ordre que dans `x`

(b) renvoie `true` si et seulement si tous les éléments de `y` apparaissent dans `x` dans le même ordre que dans `y`

(c) renvoie `true` si et seulement si tous les éléments de `y` apparaissent dans `x` dans n'importe quel ordre

(d) renvoie `true` si et seulement si tous les éléments de `y` apparaissent au moins une fois dans `x` (si `y` contient plusieurs fois le même élément, une seule apparition dans `x` est nécessaire)

(e) renvoie `true` si et seulement si tous les éléments de `x` apparaissent au moins une fois dans `y` (si `x` contient plusieurs fois le même élément, une seule apparition dans `y` est nécessaire)

2. Le test de la ligne 2 est :

(a) inutile car on a toujours `k<y.length`

(b) utile car il permet d'accélérer le traitement d'un cas particulier

(c) nécessaire au bon fonctionnement de programme qui risque de planter sinon car on ne vérifie pas que `k<y.length` avant d'accéder à `y[k]`

Exercice 2.4 :

On considère la méthode :

```
1  public static int c(int[] x,int[] y,int pos) {
2    for(int i=pos;i<x.length;i++) {
3      int j=0;
4      while(j<y.length && y[j]==x[i+j]) {
5        j++;
6      }
7      if(j==y.length) {
8        return i;
9      }
10   }
11   return -1;
12 }
```

1. Cette méthode ne fonctionne pas toujours et provoque parfois des accès en dehors des tableaux paramètres. Pour qu'elle fonctionne, il faut :
   (a) remplacer la ligne 3 par `int j=i ;`
   (b) modifier le test de la ligne 4 en `(j<y.length && i+j<x.length && y[j]==x[i+j])`
   (c) modifier le test de la ligne 4 en `(j<y.length/2 && y[j]==x[i+j])`
   (d) ajouter entre les lignes 1 et 2 l'instruction `if(x.length>y.length) return -1 ;`
2. La méthode modifiée :
   (a) renvoie le nombre d'éléments en commun entre les tableaux `x` et `y` (-1 s'il n'existe pas d'élément en commun)
   (b) renvoie la position de la première occurence du tableau `y` dans le tableau `x` à partir de la position `pos` incluse (-1 s'il n'existe pas de telle occurence)
   (c) renvoie la position de la première occurence du tableau `y` dans le tableau `x` (-1 s'il n'existe pas de telle occurence)
   (d) renvoie la position de la première occurence du tableau `y` dans le tableau `x` à partir de la position `pos` non incluse (-1 s'il n'existe pas de telle occurence)
3. On reprend la méthode **d'origine** et on modifie le test de la ligne 2 en le remplaçant par `i<x.length-y.length` :
   (a) la méthode ne plante pas et donne les mêmes résultats qu'avec la bonne modification proposée à la question (a)
   (b) la méthode ne plante pas mais ne donne pas les mêmes résultats qu'avec la bonne modification proposée à la question (a)
   (c) la méthode ne plante pas, donne les mêmes résultats qu'avec la bonne modification proposée à la question (a), et est plus efficace dans certains cas
   (d) cela n'a pas d'effet : la méthode ne fonctionne toujours pas

Exercice 2.5 :

On considère la méthode :

```
1 public static int d(int[] x,int y) {
2   for(int i=0;i<x.length;i++) {
3     if(x[i]==y) {
4       return i;
5     } else {
6       return -1;
7     }
8   return -1;
9 }
```

1. Cette méthode :
   (a) renvoie 0 si la première case de `x` contient `y` et -1 sinon
   (b) renvoie la liste des numéros des cases de `x` contenant `y` ou -1 si `y` n'apparaît pas dans `x`
   (c) renvoie la position de la première occurence de `y` dans `x` ou -1 si `y` n'apparaît pas dans `x`
2. Modifier cette méthode pour qu'elle compte le nombre de cases de `x` qui contiennent la valeur `y` et renvoie ce nombre.
3. On modifie la méthode **d'origine** en remplaçant la ligne 2 par la ligne :
   `for(int i=1 ;i<x.length ;i++) {`
   La nouvelle méthode :
   (a) compile et renvoie un résultat parfois différent de la version non mondifiée
   (b) ne compile pas
   (c) compile et fonctionne comme avant la modification
   (d) compile et renvoie toujours 0

Exercice 2.6 :

On considère la méthode :

```
1 public static int e(int[] x,int[] y) {
2   int i=0;
3   for(;i<x.length;i++) {
4     if(x[i]==y[i]) {
5       break;
6     }
7   }
8   return i;
9 }
```

1. Cette méthode :
   (a) renvoie le nombre de cases contenant la même valeur dans `x` que dans `y`
   (b) ne compile pas
   (c) renvoie la position de la première case de `x` contenant la même valeur que la case correspondante de `y`, et `x.length` si une telle case n'existe pas
   (d) renvoie la position de la première case de `x` contenant la même valeur que la case correspondante de `y`, et 0 si une telle case n'existe pas

2. Modifier cette méthode pour qu'elle fonctionne correctement même si `y` désigne un tableau plus court que `x`. La méthode obtenue doit compiler et ne jamais planter, quelles que soient les longueurs des tableaux paramètres.

Exercice 2.7 :

On considère la méthode :

```
1 public static int g(int[] x) {
2   int t=0;
3   for(int i=1;i<x.length;i++) {
4     if(x[t]<x[i]) {
5       t=i;
6     }
7   }
8   return t;
9 }
```

1. Cette méthode :
   (a) renvoie la valeur du plus grand élément de `x`
   (b) renvoie la position de la dernière occurence du plus petit élément de `x`
   (c) renvoie la position de la dernière occurence du plus grand élément de `x`
   (d) renvoie la valeur du plus petit élément de `x`
   (e) renvoie la position de la première occurence du plus grand élément de `x`
   (f) renvoie la position de la première occurence du plus petit élément de `x`
2. On appelle la méthode avec comme paramètre le tableau `{4,4,4}`. La méthode renvoie :
   (a) 4
   (b) 0
   (c) 3
3. On remplace la condition `x[t]<x[i]` par `x[t]<=x[i]`. Quelle est la conséquence de cette modification :
   (a) aucune conséquence
   (b) si le plus petit élément de `x` apparaît dans plusieurs cases, la méthode renvoie la position de la première occurence de cet élément
   (c) si le plus grand élément de `x` apparaît dans plusieurs cases, la méthode renvoie la position de la dernière occurence de cet élément
   (d) si le plus petit élément de `x` apparaît dans plusieurs cases, la méthode renvoie la position de la première occurence de cet élément
   (e) si le plus grand élément de `x` apparaît dans plusieurs cases, la méthode renvoie la position de la dernière occurence de cet élément

Exercice 2.8 :

On considère la méthode :

```
1 public static boolean h(int[] x,int[] y) {
2   for(int i=0;i<x.length;i++) {
3     if(x[i]==y[i]) {
4       return true;
5     } else {
6       return false;
7     }
8   }
9 }
```

1. La méthode ne compile pas car :
   (a) il manque un `return`
   (b) il est interdit d'avoir plusieurs `return` dans une même boucle
   (c) il manque des accolades
   (d) on ne vérifie pas que les deux tableaux sont de même taille
2. Donner une version de cette méthode qui compile (avec le moins de modifications possibles).

Exercice 2.9 :

On considère la méthode :

```
1  public static boolean i(int[] y,int[] x) {
2    for(int i=0;i<y.length;i++) {
3      int j=0;
4      while(j<x.length && x[j]!=y[i]) {
5        j++;
6      }
7      if(j==x.length) {
8        return false;
9      }
10   }
11   return true;
12 }
```

1. Cette méthode :
   (a) renvoie `true` si et seulement si tous les éléments de `y` apparaissent dans `x` dans n'importe quel ordre
   (b) renvoie `true` si et seulement si tous les éléments de `y` apparaissent dans `x` dans le même ordre que dans `y`
   (c) renvoie `true` si et seulement si tous les éléments de `y` apparaissent au moins une fois dans `x` (si `y` contient plusieurs fois le même élément, une seule apparition dans `x` est nécessaire)
   (d) renvoie `true` si et seulement si tous les éléments de `x` apparaissent dans `y` dans le même ordre que dans `x`

(e) renvoie `true` si et seulement si tous les éléments de `x` apparaissent au moins une fois dans `y` (si `x` contient plusieurs fois le même élément, une seule apparition dans `y` est nécessaire)

2. On remplace le test `(j<x.length && x[j] !=y[i])` par le test `(x[j] !=y[i] && j<x.length)`. La nouvelle version de la méthode :
   (a) plante toujours
   (b) plante dans les cas où elle devrait renvoyer `false`
   (c) ne compile pas
   (d) fonctionne parfaitement
   (e) plante dans les cas où elle devrait renvoyer `true`

Exercice 2.10 :

On considère la méthode :

```
1  public static boolean j(int[] x) {
2    int p=0;
3    int k=0;
4    while(2*k<x.length) {
5      if(x[2*k+1]==x[2*k]) {
6        p++;
7      }
8      k++;
9    }
10   return p==x.length/2;
11 }
```

1. Cette méthode ne fonctionne que si le tableau paramètre est :
   (a) de longueur impaire
   (b) de longueur paire
2. On modifie la condition `x[2*k+1]==x[2*k]` pour que la méthode fonctionne pour tout tableau. Il faut utiliser la condition :
   (a) `x[2*k+1]==x[2*k] && 2*k+1<x.length`
   (b) `2*k+1<x.length && x[2*k+1]==x[2*k]`
   (c) `2*k<x.length && x[2*k+1]==x[2*k]`
3. La méthode corrigée :
   (a) renvoie `true` si et seulement si toutes les cases de numéro impair contiennent la même valeur
   (b) renvoie `true` si et seulement si toutes les cases de numéro pair contiennent la même valeur
   (c) renvoie `true` si et seulement si chaque case de numéro impair contient la même valeur que la case qui la suit (si elle existe)
   (d) renvoie `true` si et seulement si chaque case de numéro pair contient la même valeur que la case qui la suit (si elle existe)
4. En utilisant une boucle `for` et un `return` dans la boucle, écrire une version plus simple de cette méthode.

Exercice 2.11 :

On considère la méthode :

```
1  public static void a(double[] t) {
2    for(int i=1;i<t.length;i++) {
3      if(t[i-1]>t[i]) {
4        double tmp=t[i];
5        t[i]=t[i-1];
6        t[i-1]=tmp;
7      }
8    }
9  }
```

1. On transmet à la méthode un tableau de longueur 1. La méthode :
   (a) plante
   (b) ne change rien au tableau
2. Quel est l'effet de cette méthode sur son tableau paramètre :
   (a) elle place en dernière position le plus grand élément du tableau
   (b) elle place en dernière position le plus petit élément du tableau
   (c) elle trie le tableau par ordre croissant
   (d) elle ne change rien au tableau
   (e) elle trie le tableau par ordre décroissant

Exercice 2.12 :

On considère la méthode :

```
1  public static int b(double[] x,double[] y) {
2    int k=0;
3    for(int i=0;i<x.length;i++) {
4      for(int j=0;j<y.length;j++) {
5        if(x[i]==y[j]) {
6          k++;
7        }
8      }
9    }
10   return k;
11 }
```

1. Cette méthode renvoie :
   (a) le nombre d'éléments de `y` qui apparaissent dans `x` (à n'importe quel emplacement)
   (b) le nombre de cases de `x` dont le contenu est égal à celui de la case de même indice dans `y`
   (c) la somme du nombre de fois que chaque élément de `x` apparaît dans `y`
   (d) le nombre d'éléments de `x` qui apparaissent dans `y` (à n'importe quel emplacement)

2. Dans la ligne 4 de la méthode d'**origine**, on remplace `int j=0` par `int j=i`. Quand on tente de l'appeler avec un tableau `x` **plus court** que `y`, la nouvelle méthode :
   (a) plante
   (b) renvoie un résultat inférieur ou égal à celui qu'aurait renvoyé l'ancienne méthode
   (c) renvoie un résultat strictement inférieur à celui qu'aurait renvoyé l'ancienne méthode
   (d) renvoie le même résultat que l'ancienne méthode
3. Dans la ligne 4 de la méthode d'**origine**, on remplace `int j=0` par `int j=i`. Quand on tente de l'appeler avec un tableau `x` **plus long** que `y`, la nouvelle méthode :
   (a) renvoie le même résultat que l'ancienne méthode
   (b) renvoie un résultat strictement inférieur à celui qu'aurait renvoyé l'ancienne méthode
   (c) renvoie un résultat inférieur ou égal à celui qu'aurait renvoyé l'ancienne méthode
   (d) plante

Exercice 2.13 :

On considère la méthode :

```
1  public static int c(double[] t,double x) {
2    int i=0;
3    int j=t.length-1;
4    while(i<=j) {
5      int m=(i+j)/2;
6      if(t[m]==x) {
7        return m;
8      }
9      if(t[m]<x) {
10       i=m+1;
11     } else {
12       j=m-1;
13     }
14   }
15   return -1;
16 }
```

1. Pourquoi l'entier `m` désigne-t-il toujours un indice valide pour les cases du tableau `t` ?
2. Pourquoi la boucle s'arrête-t-elle toujours ?
3. On appelle la méthode avec comme paramètre `t` un tableau trié par ordre croissant. La méthode :
   (a) renvoie la position de la dernière occurrence de `x` dans `t` et -1 si `x` n'apparaît pas dans `t`
   (b) renvoie la position d'une occurrence de `x` dans `t` et -1 si `x` n'apparaît pas dans `t`
   (c) renvoie la position de la première occurrence de `x` dans `t` et -1 si `x` n'apparaît pas dans `t`
   (d) renvoie la liste des positions des occurrences de `x` dans `t` et -1 si `x` n'apparaît pas dans `t`

4. On appelle la méthode avec comme paramètre un tableau `t` de longueur 16. Combien de fois la boucle s'exécute-t-elle au maximum ? Donner un exemple de tableau et de valeur numérique pour lesquels le maximum est atteint.
5. On appelle la méthode avec comme paramètre `t` un tableau non trié. La méthode :
   (a) plante
   (b) renvoie la position d'une occurence quelconque de `x` dans `t`
   (c) renvoie parfois la position d'une occurence quelconque de `x` dans `t`, parfois -1
   (d) renvoie toujours -1

2.2 Programmation

Exercice 2.14 :

Ecrire l'algorithme du calcul de la moyenne pondérée d'une liste de réels. On supposera donnée une liste de réels représentant les pondérations à appliquer.

Exercice 2.15 :

Ecrire une méthode qui détermine si un tableau est "inclus" dans un autre. Un tableau `a` est "inclus" dans un tableau `b` si `a` est plus court que `b` (au sens large), et si tous les éléments de `a` apparaissent dans `b` dans le même ordre que dans `a`. Par exemple le tableau `{1,-4}` est inclus dans `{4,1,0,-4,8}` , mais n'est pas inclus dans `{-4,1,0}` .

Exercice 2.16 :

Ecrire une méthode `boolean suivant(int[] t,int fin)` qui donne à `t` la valeur suivante, au sens d'un compteur kilométrique : si l'élément `t[0] < fin` alors `t[0]` augmente de 1 sinon il passe à 0 et on refait la même chose avec `t[1]`, etc. Si tous les éléments valent `fin` alors ils passent tous à 0 et la méthode renvoie false. Dans les autres cas elle renvoie true.

Exercice 2.17 :

Pour trouver tous les nombres premiers inférieurs à un entier `N`, on procède comme suit (il s'agit de l'algorithme du crible d'Eratosthène) :
– On range par ordre croissant, dans une liste, les nombres entiers compris entre 1 et `N` ;
– on "barre" tous les multiples stricts de 2 c'est-à-dire tous les entiers multiples de 2, autres que 2 lui-même ;
– on considère alors l'entier suivant, s'il est non barré, on barre tous ses multiples stricts ;
– on répète cette opération jusqu'à la fin de la liste.

La liste des `N` premiers entiers et la propriété pour chacun d'eux d'être barré ou non, sera représentée par un tableau booléen `liste` indicé de 1 à `N` de telle sorte que `liste[i]` vaut `true` si i est "barré", faux sinon (pour simplifier, on ne considérera pas la case d'incide 0).

Le tableau `liste` sera initialisé de telle sorte que chacun de ses éléments soit égal à `false` (i.e. non barré).

1. Appliquez cette méthode "à la main" à la liste des entiers inférieurs à 20 en faisant apparaître à chaque étape le diviseur choisi et la liste des nombres barrés.
2. Ecrivez un programme qui détermine la liste des nombres premiers inférieurs à `N` (valeur saisie par l'utilisateur) en suivant la méthode qui vient d'être proposée.

Exercice 2.18 :

Ecrivez une méthode qui à deux tableaux d'entiers de même longueur `a` et `b` associe le tableau contenant `b[i]` fois de suite la valeur contenue dans la case `a[i]` pour tout `i`. On suppose que les éléments de `b` sont tous positifs ou nuls. Le résultat de la méthode pour `a={1,-2,2}` et `b={1,2,1}` sera donc le tableau `{1,-2,-2,2}` (c'est-à-dire 1 fois 1, 2 fois -2 et 1 fois 2).

Exercice 2.19 :

On souhaite calculer une moyenne mobile pour lisser un tableau. On suppose donné un tableau $u = (u_0, u_1, \ldots, u_{n-1})$ et on souhaite lui associer un nouveau tableau $v = (v_0, v_1, \ldots, v_{n-1})$. L'application d'une moyenne mobile d'ordre k définit v_p comme :

$$v_p = \frac{1}{2k-1} \sum_{i=p-k+1}^{p+k-1} u_i$$

Pour une moyenne d'ordre 2, il s'agit donc pour obtenir v_3 de calculer la moyenne de u_2, u_3 et u_4, etc.

1. Certains termes v_p ne sont pas calculables : indiquer lesquels (en fonction de k).
2. Programmer une méthode qui à un tableau u et un entier k associe le tableau v obtenu par moyenne mobile d'ordre k. On fixera arbitrairement les v_p non calculables à 0.

2.3 Tableaux et tri

Exercice 2.20 :

Le but de cet exercice est de réaliser une méthode qui permet de trier un tableau d'entiers sur place (c'est-à-dire en le modifiant). Voici l'algorithme de la stratégie de tri retenue (stratégie dite par sélection) :

Données :
– l'objet `tab` représentant la liste $u = (u_0, u_1, \ldots, u_{n-1})$.

Résultat : l'objet `tab` représentant la liste $v = (v_0, \ldots, v_{n-1})$ dont les éléments sont les mêmes que ceux de u mais triés par ordre croissant.

1. répéter pour `i` allant de 0 à $n-1$:
   (a) trouver le plus petit élément du tableau `tab` **en commençant au rang `i` inclus**.
   (b) échanger le plus petit élément avec l'élément de rang `i`.
2. **Résultat** : le contenu de `tab`

Questions :

1. Faire tourner l'algorithme à la main (en indiquant le contenu de l'objet `tab` **après** chaque itération de la boucle) sur les deux tableaux suivants : $(0, 3, 2, 7, 4)$ et $(5, -2, 3, 4, 1)$.
2. Écrire une méthode `posMin` qui à un tableau d'entiers `t` et à un entier `p` associe le rang du plus petit élément de `t` de rang **supérieur ou égal** à `p` (on supposera que `p` est inférieur ou égal à l'indice maximal utilisable pour le tableau `t`).
3. On considère que la méthode `posMin` est appelée avec un tableau de longueur `n` et avec comme paramètre entier `p`. Combien de comparaisons cette méthode effectue-t-elle pour calculer la position du minimum (à exprimer en fonction de `n` et `p`) ?

4. Écrire une méthode `tri` qui modifie le tableau `tab` qui lui est transmis en paramètre afin de le trier. Elle utilisera la méthode `posMin`.
5. On suppose que la méthode `tri` est appelée avec un tableau de longueur `n`. Combien de comparaisons effectue-t-elle en tout pour trier ce tableau (à exprimer en fonction de `n`) ?

Exercice 2.21 :

Écrire une méthode `void mélange(int[] t)` qui modifie l'ordre des éléments de `t` de façon aléatoire. Après l'appel, le tableau `t` doit contenir exactement les mêmes valeurs qu'au départ, mais dans un ordre différent.

Exemple : si `t` désigne le tableau `{1,2,3,4}` , après l'appel `mélange(t)`, le contenu de `t` peut être `{2,4,3,1}` (par exemple).

Indication : on peut créer un objet `Random` grâce à la construction `Random rng=new Random() ;`. Ensuite, chaque appel `rng.nextInt(n)` renvoie un entier (`int`) choisi aléatoirement entre 0 (inclus) et `n-1` (inclus).

Exercice 2.22 :

Ecrire une méthode de classe `double[] kPlusPetit(double[] t,int k)` qui à un tableau `t` et un entier `k` associe un tableau contenant les `k` plus petits éléments de `t`, triés par ordre croissant. Par exemple, le résultat de `kPlusPetit(new double[] {1.5,-2,7,2.5,6,-3},2)` est le tableau `{-3,-2}` . On ne souhaite pas trier le tableau `t` qui ne doit pas être modifié par la méthode.

Il est vivement conseillé de commencer par écrire une méthode `void insertion(double[] r,double u)` qui insère un élément `u` dans un tableau `r` trié par ordre croissant. Plus précisément, si `u` est plus grand que le plus grand élément de `r`, on ne fait rien. Sinon, on supprime le plus grand élément de `r` et on insère `u` dans le tableau `r` de sorte qu'après l'appel de la méthode, `r` soit toujours trié par ordre croissant. Par exemple, si `r` fait référence au tableau `{-2,1.5}` , après l'appel de `insertion(r,-3)`, le contenu de `r` est devenu `{-3,-2}` .

Exercice 2.23 :

Ecrire une méthode qui à un tableau d'entiers **trié** (par ordre croissant) et un entier, associe la position de cet entier dans le tableau (et `-1` si l'entier n'apparaît pas dans le tableau). On utilisera une recherche dichotomique. On compare l'élément à chercher avec le milieu du tableau. S'il est plus petit, on recommence sur le demi-tableau de gauche ; s'il est plus grand, on recommence avec le demi-tableau de droite ; sinon on a trouvé la position.

3 Tableaux et chaînes de caractères

3.1 Programmation

Exercice 3.1 :

Écrire une méthode `String[] début(String[] t,String u)` qui à `t` et `u` associe le tableau des chaînes de caractères contenues dans `t` et qui commencent par `u`. La méthode doit conserver l'ordre des chaînes dans `t`.

Exemple : si `t` désigne le tableau `{"toto","TOTO","toTI"}` et si `u` désigne la chaîne `"to"`, le résultat de `début(t,u)` est le tableau `{"toto","toTI"}` .

Exercice 3.2 :

Programmer une méthode `String[] tri(String[] t)` qui trie le tableau `t` en utilisant l'algorithme de votre choix. Après l'appel, le tableau `t` doit être **inchangé** alors que le tableau résultat doit contenir les mêmes éléments que `t`, mais rangés en ordre croissant.

Indication : on rappelle que la classe `String` définit une méthode **d'instance** `int compareTo(String t)` qui compare la chaîne appelante avec la chaîne `t`. Le résultat est strictement négatif si la chaîne appelante se range avant `t` dans le dictionnaire, strictement positif dans le cas contraire (une valeur nulle est renvoyée quand les deux chaînes sont égales).

3.2 Problèmes

Problème 3.3 :

Dans tout le problème, si `s` est une chaîne de caractères, $\mathtt{s}_k$ désigne son caractère numéro k.

Manipulation de chaînes de caractères

1. Ecrire une méthode `dup` prenant comme paramètre un entier `k` et une chaîne de caractères `s` et renvoyant comme résultat une nouvelle chaîne `t` définie de la façon suivante : $\mathtt{t}_i = \mathtt{s}_i$ pour tout i inférieur strictement à `n` la longueur de `s`, et $\mathtt{t}_i = \mathtt{s}_{i-\mathtt{n}}$ pour i compris au sens large entre `n` et `n+k-1`. Si `k` est trop grand ou trop petit, la méthode renvoie la chaîne `s`.
   Exemple : l'appel `dup(2,"abcd")` renvoie la chaîne `"abcdab"`.
2. Ecrire une méthode `cut` qui à un entier `k` et une chaîne de caractères `s` associe la chaîne de caractères constituée des `n-k` premiers caractères de `s` (où `n` désigne la longueur de `s`).
   Exemple : l'appel `cut(2,"abcd")` renvoie la chaîne `"ab"`.
3. Ecrire une méthode `rotateLeft` qui à un entier `k` et une chaîne de caractères `s` de longueur `n` associe la chaîne de même longueur `t` dont les caractères sont dans l'ordre $(\mathtt{s}_k, \mathtt{s}_{k+1}, \ldots, \mathtt{s}_{n-1}, \mathtt{s}_0, \mathtt{s}_1, \ldots, \mathtt{s}_{k-1})$.
   Exemple : l'appel `rotateLeft(2,"abcde")` renvoie la chaîne `"cdeab"`.
4. Ecrire une méthode `rotateRight` qui à un entier `k` et une chaîne de caractères `s` de longueur `n` associe la chaîne de même longueur `t` dont les caractères sont dans l'ordre $(\mathtt{s}_{n-k}, \mathtt{s}_{n-k+1}, \ldots, \mathtt{s}_{n-1}, \mathtt{s}_0, \mathtt{s}_1, \ldots, \mathtt{s}_{n-k-1})$.
   Exemple : l'appel `rotateRight(2,"abcde")` renvoie la chaîne `"deabc"`.
5. Ecrire une méthode `mirror` qui à une chaîne de caractères `s` associe son "miroir", c'est-à-dire la chaîne constituée des mêmes caractères que `s` mais dans le sens contraire.
   Exemple : l'appel `mirror("abcd")` renvoie la chaîne `"dcba"`.

Chaîne de commande

1. On considère une chaîne caractères constituée exclusivement de chiffres et du caractère ','. Plus précisément, on suppose que la chaîne est une liste d'entiers positifs, chaque nombre étant séparé du précédent par une virgule.
   Exemple : la chaîne `"12,3,4,25,13"`.
   Ecrire une méthode `parse` qui à une chaîne de caractères du format qui vient d'être décrit associe un tableau contenant les entiers donnés dans la chaîne (dans l'ordre de leur apparition dans la chaîne). On supposera donnée une méthode `toInt` dans la classe `Convert` qui à une chaîne de caractères représentant un entier associe la valeur de cet entier sous forme d'une valeur de type `int`.
   Exemple : l'appel `parse("12,3,4,25,13")` renvoie le tableau `{12,3,4,25,13}` .

2. On associe un code aux cinq opérations programmées dans les questions précédentes :

code	méthode
0	`dup`
1	`cut`
2	`rotateLeft`
3	`rotateRight`
4	`mirror`

Ecrire une méthode `applique` prenant comme paramètre un tableau d'entiers `t` et une chaîne de caractères `s`. La méthode va appliquer à la chaîne `s` les opérations décrites dans `t`. Plus précisément, elle va lire le premier entier dans `t`. Si cet entier est compris (au sens large) entre 0 et 3, elle appelle la méthode correspondante avec comme paramètre le second entier et la chaîne `s`, puis place le résultat dans `s`. Si l'entier vaut 4, elle place dans `s` le miroir de `s`. La méthode a donc utilisé 1 ou 2 entiers du tableau. Elle continue ensuite avec le reste du tableau et renvoie finalement le contenu de la chaîne `s`.

Exemple : on considère l'appel de `applique` avec comme paramètres la chaîne `"abcdef"` et le tableau `{1,1,2,4,4,0,2}` . Comme le premier entier du tableau est 1, on commence par appeler la méthode `cut(1,"abcdef")` (le paramètre 1 est le deuxième élément du tableau). On obtient ainsi la chaîne `"abcde"`. Le prochain élément du tableau est alors 2, suivit de 4. On appelle donc `rotateLeft(4,"abcde")`, ce qui nous donne la chaîne `"eabcd"`. On applique ensuite la méthode `mirror` (car l'élément suivant du tableau est 3), ce qui donne la chaîne `"dcbae"`. On termine les opérations par un appel à la méthode `dup(2,"dcbae")`, ce qui nous donne la chaîne `"dcbaedc"`, résultat de l'appel à `applique`.

3. si la suite d'opérations décrite par un tableau ne contient pas l'opération `cut`, elle est réversible, c'est-à-dire qu'il existe une suite d'opérations (contenant éventuellement l'opération `cut`) qui permet de revenir à la chaîne d'origine en partant de la chaîne résultat. Ecrire une méthode qui à un tableau d'opérations associe la suite inverse (quand c'est possible).

Exemple : on considère la suite `{2,4,4,0,2}` . La dernière opération ajoute deux caractères à la fin de la chaîne. Pour annuler cette opération, il suffit donc d'appeler la méthode `cut(s,2)`. L'opération précédente réalise un miroir. Pour l'annuler, il suffit de réaliser un miroir. Enfin, la première opération est une rotation vers la gauche de 2, et il suffit donc d'effectuer une rotation vers la droite de 2 pour l'annuler. La suite inverse est donc `{1,2,4,3,2}` .

Problème 3.4 :

Dans cet exercice, on note a_k le k-ième caractère de a (numérotés à partir de 0).

Questions :

1. On dit qu'une chaîne a est un préfixe de la chaîne b si le début de b est exactement formé par les caractères de a. Ecrire une méthode qui à deux chaînes de caractères associe `true` si la première est un prefixe de la seconde et `false` sinon.
2. On dit qu'une chaîne a est une sous-séquence de la chaîne b si toutes les lettres de a apparaissent dans b dans l'ordre. La chaîne *baba* est par exemple une sous-séquence de *cbbacbca*, mais pas de *abab* (qui contient les mêmes lettres, mais pas dans le même ordre).
   (a) Ecrire une méthode qui à une chaîne a, un caractère c et un entier i associe le plus petit entier j tel que $j \geq i$ et $a_j = c$. S'il n'existe pas de tel entier, la méthode renvoie la longueur de la chaîne a.

(b) Utiliser la méthode précédente pour programmer une méthode qui à deux chaînes a et b associe `true` si a est une sous-séquence de b. Il suffit de chercher la première occurence de a_0 dans b, puis celle de a_1 dans b, à partir de la place de a_0, etc.

3. On considère deux chaînes de caractères a et b. On définit le tableau `t` des plus longues sous-séquences de la façon suivante :
   – si a est de longueur m et b de longueur n, le tableau `t` possède deux dimensions : $m+1$ lignes et $n+1$ colonnes ;
   – si `i=0` ou `j=0`, on pose `t[i,j]=0` ;
   – si `i>0` et `j>0` et $a_{\texttt{i-1}} \neq b_{\texttt{j-1}}$, on pose `t[i,j]=max(t[i,j-1],t[i-1,j])` ;
   – si `i>0` et `j>0` et $a_{\texttt{i-1}} = b_{\texttt{j-1}}$, on pose `t[i,j]=1+t[i-1,j-1]`.
   On voit donc que pour calculer `tab`, on doit d'abord remplir la première ligne et la première colonne, puis pour chaque ligne, remplir colonne par colonne. Ecrire une méthode qui à deux chaînes de caractères a et b associe le tableau `tab` qui vient d'être défini.