アルゴリズムとデータ構造 2019 第６回　　年　　組　　番 氏名

1. 下記は，最も単純な**文字列探索**アルゴリズムである。このメソッドは，文字列 text の中に文字列 word が含まれるか先頭から調べ，最初に見つかった位置（0 以上）を返す。ただし，見つからなかった場合には -1 を返す。空欄を適切に埋めて実行し，動作を確認せよ。このアルゴリズムがなぜ非効率かも考察せよ。

```
public static int findString(String text, String word) {

    int tlen = text.length();
    int wlen = word.length();

    // text の中の位置 t から word が入っているか調べる（残りが wlen 文字以上なら続ける）
    for (int t = 0; t <=               ; t++) {

        // その位置から word の文字が全部入っているか調べる（w は一致文字数）
        int w;
        for (w = 0; w <        ; w++) {

            if (text.charAt(        ) != word.charAt(        ))  /* ★ */
                break;
        }
        if (w == wlen) return t;
    }

    return
}
```

2. 下記のクラスについて問いに答えよ（このクラスには適当なメソッドを追加してもよい）。

```
public class Item {        // 商品
    public int code;       // 品番
    public String name;    // 品名
}
```

この Item の配列に対して，挿入ソートを用いて商品を品番順に整列するメソッドと，2 分探索を用いて引数の（仮の）商品データと同じ品番の商品を探し出すメソッドを作成し，動作を検証せよ。探索で見つからなかった場合は負数を返すようにせよ。以下にメソッドの名前と引数を示す。

```
public static void sort(Item[] data)
public static int binarySearch(Item[] data, Item key)
```

3. Java ではクラスに Comparable インタフェースを実装すると比較可能（順序付け可能）になる。これによる順序を「自然な順序」という。2.の Item を以下のように品番によって順序が付くように修正せよ。

```
public class Item implements Comparable<Item> {  // Item が比較可能であることを示す
    // 2.で定義した部分はそのまま

    @Override
    public boolean equals(Object obj) { /* プロ II の教科書 14.2.6 を参考に実装 */ }

    @Override
    public int compareTo(Item item2) { /* code の小さい順になるように実装 */ }

    @Override
    public int hashCode() { return this.code; }  // equals をオーバーライドするとき適切に定義
}
```

Java には配列操作のためのクラス java.util.Arrays があり，自然な順序によるソートには静的メソッド Arrays.sort（プロ II 教科書 6.5.1 参照），同じく 2 分探索には Arrays.binarySearch（各自調べること）が利用できる。修正した Item とこれらによって，2.と同じ結果を得るプログラムを作成せよ。

4. 配列を「自然な順序」以外の順序を用いて整列・探索したい場合には，まず要素同士を比較するために，java.util.Comparator インタフェースを実装した比較方法を表すクラスを作る。以下に例を示す。

```
class ItemComparator implements Comparator<Item> {  // Item のための比較方法のクラスの実装例
    @Override
    public int compare(Item a, Item b) {  // 比較メソッド（a<b なら負，a==b なら 0，a>b なら正）
        return b.code - a.code;           // この例では 3.で定義した「自然な順序」の逆順になる
    }
}
```

そして，Arrays.sort(items, new ItemComparetor()) などとして，比較用クラスのインスタンスをメソッドの最後の追加引数として渡せばよい。この機能を利用して，Item の配列を品番ではなく，品名で並べ替え，品名で探索するプログラムを作成せよ。

5. 普通の配列は作成時のサイズ（要素数）を変更することができないが，それが不便なことも多い。そこで，Java では，java.util パッケージに ArrayList<E>（E には要素のクラス名が入る）という，サイズが動的に変更できる配列（のようなクラス）が用意されている。

ここで，<E>は**ジェネリクス（総称型）**と呼ばれる記法で，E の部分にはインスタンス生成時に任意のクラス名を指定できる。ただし，int, double などの基本型は，Integer, Double などの対応するラッパークラス（プロ II 教科書 14.3 参照）で代用する。以下に，ArrayList の使用パターンをいくつか例示する。

必要な import 文

```
import java.util.ArrayList;
```

動的配列の生成（クラス）

```
ArrayList<String> strArray = new ArrayList<String>();
strArray.add("abc");                        // 末尾に要素を追加するのは.add を使う（配列の長さは自動的に拡張）
```

動的配列の使用例（基本型）

```
ArrayList<Integer> intArray = new ArrayList<Integer>();
Integer a = new Integer(32);                // クラス版の整数（Integer クラス）のインスタンスを生成する
Integer b = 32;                             // 自動変換（autoboxing）機能を使えば基本型を直接代入できる
intArray.add(a);
intArray.add(64);                           // 引数に直接基本型を渡した場合も自動変換（autoboxing）される
```

配列要素へのアクセス

```
int n = intArray.size();                    // 要素数を得るには.size()を使う（通常の配列の.length と同じ）
for (int i = 0; i < n; i++) {
    int c = intArray.get(i);                // i 番目の要素を取得するには.get(i)とする
    System.out.println(c);                  // 上の例は，自動変換（auto-unboxing）で基本型の c に直接代入
}
strArray.set(0, "xyz");                     // i 番目の要素を x に入れ替えるには.set(i, x)とする
for (String str : strArray)                 // 通常の配列のように，拡張 for 文も使える
    System.out.println(str);
```

配列要素の探索

```
int j = intArray.indexOf(64);               // 先頭から線形探索でデータを探す
int k = intArray.lastIndexOf(32);           // 末尾から線形探索でデータを探す
```

これらの例を参考にして，2.で作成したプログラムを普通の配列ではなく ArrayList<Item>を利用するように書き換え，動作を検証せよ。ArrayList の詳細についてはマニュアル等を参照すること。

```
public static void sort(ArrayList<Item> data)
public static int binarySearch(ArrayList<Item> data, Item key)
```

6. ArrayList に対して，Arrays.sort や Arrays.binarySearch は使えないが，代わりに java.util.Collections の静的メソッドである Collections.sort および Collections.binarySearch をまったく同じように使える。3.と 4.で作成したプログラムを ArraysList<Item>を利用するように書き換え，動作を確認せよ。

7. 【発展】以下のプログラムは，効率的な文字列探索アルゴリズムであるボイヤー・ムーア法（簡易版）の例である。このアルゴリズムの特徴は，探索文字列の後ろから文字比較を行うことによって，不一致だった場合に探索位置を大きくスキップできることである。このアルゴリズムと 1.のアルゴリズムの効率を比較するために，文字比較の実行位置である/* ★ */の部分の実行回数を数えて最後に表示するように両方のプログラムを改造し，テキストや探索文字列を使って文字比較の回数を比較してみよ。

```
public class Program {

    // 日本語にも対応するため，マスクをかけて文字コードの下位 8 ビットだけ利用する
    final static int BITMASK = 0xff;

    // あらかじめ，テキストと探索文字列を比較中に，照合された文字が異なっていた場合，
    // テキスト側の文字から判断して，探索位置を何文字スキップできるか表にしておく
    public static int[] makeSkipTable(String pattern) {
        int len = pattern.length();
        int[] skipTable = new int[BITMASK + 1];  // 256 文字分の表

        // 探索文字列に含まれない文字の場合，探索文字列の長さの分だけ探索位置をスキップ
        for (int i = 0; i <= BITMASK; i++)
            skipTable[i] = len;

        // 探索文字列に含まれる文字の場合，その文字の位置が合うように探索位置をスキップ
        for (int i = 0; i < len - 1; i++) {
            int code = pattern.charAt(i) & BITMASK;
            skipTable[code] = len - i - 1;
        }

        return skipTable;
    }

    // ボイヤー・ムーア法（簡易版）による文字列探索
    public static int findStringBM(String text, String word) {

        // あらかじめ探索文字列からスキップ表を作成して準備しておく
        int[] skipTable = makeSkipTable(word);
        int tlen = text.length();
        int wlen = word.length();

        int t = 0;
        while (t <= tlen - wlen) {
            // テキストと探索文字列を後ろから 1 文字ずつ照合していく
            int w;
            for (w = wlen - 1; w >= 0; w--) {
                // テキストから照合位置の文字を取り出し，探索文字列と比較する
                int code = text.charAt(t + w);
                if (code != word.charAt(w)) {  /* ★ */
                    // 文字が合わなかった場合，その文字からスキップ量を調べる
                    t += skipTable[code & BITMASK];
                    break;
                }
            }
            if (w < 0) return t;
        }
        return -1;
    }

    public static void main(String[] args) {
        String text = "Tamagawa Gakuen Kogakubu";
        System.out.println(findStringBM(text, "gawa"));
        System.out.println(findStringBM(text, "ken"));
        System.out.println(findStringBM(text, "gaku"));
        System.out.println(findStringBM("Ｔａｍａｇａｗａ　Ｇａｋｕｅｎ", "ｇａｗａ"));
    }
}
```