アルゴリズムとデータ構造 2024 第６回 演習課題 「文字列探索とジェネリクス」

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 key 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。見つからなかった場合は-1 を返す。空欄を適切に埋めて実行し，動作を確認せよ。（補足：charAt は文字列の中の指定位置の 1 文字を返す）

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

    int n = text.length();
    int m = key.length();

    // text の中で位置をずらしながら key が含まれるか調べる（text の残りが m 文字以上あれば継続）
    for (int i = 0; i <=               ; i++) {

        //  text の中の位置 i から順に，key の中の文字と同じであるか調べる（k は key の中の位置）
        int k;
        for (k = 0; k <        ; k++) {

            if (text.charAt(        ) != key.charAt(        ))  /* ★ */
                break;
        }
        if (k == m) return i;  // key の全文字（m 文字）が一致したら発見
    }
    return
}
```

2. String クラスのメソッド indexOf を用いて，「government of the people, by the people, for the people」という文字列の中から「people」を探索して全ての出現位置（インデックス）を表示し，次に「the」を探索して全ての出現位置を表示するプログラムを作成せよ。

3. 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（-1）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)  // key は探索値を入れた仮のデータ
```

4. 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（プロ II 教科書 16.1.1 参照）。
   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 16.1.3 参照）で代用する。
   次ページの表などを参考にして，文字列を要素とする ArrayList の中身を逆順に入れ替えるメソッド reverse を，for 文を使って作成し，動作を検証せよ。

```
public static void reverse(ArrayList<String> a)
```

	通常の配列	ArrayList
必要な import	なし	`import java.util.ArrayList`
配列の生成	`double [] a = new double[10]` 指定の要素数で生成される	`ArrayList<Double> a = new ArrayList<Double>()` 要素がない空の状態で生成される
要素数	`a.length`	`a.size()`
空の判定	`a.length != 0`	`a.isEmpty()`
要素の追加（末尾）	要素数は変更できない	`a.add(x)`
要素の取得	`double x = a[i]`	`double x = a.get(i)`
要素の変更	`a[i] = x`	`a.set(i, x)`
拡張 for 文	`for (double x : a)`	`for (double x : a)`
先頭から線形探索		`int i = a.indexOf(x);`
末尾から線形探索		`int i = a.lastIndexOf(x);`

5. 【発展】配列の整列や探索は自作するよりも，java.util.Arrays の機能を使う方が便利である。そのためには，要素のクラスに java.util.Comparable インタフェースを実装し，比較による順序関係を定義する必要がある。これを「自然な順序」という。まず，2.の Item を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 と矛盾しないように適切に定義
}
```

次に，この Item の配列に対して Arrays.sort（プロ II 教科書 6.6.1 参照）と Arrays.binarySearch（各自調べること）という静的メソッドを適用させて，2.と同じ結果を得るプログラムを作成せよ。

6. 【発展】配列を「自然な順序」以外の方法で整列・探索したい場合，まず，要素同士の新しい比較方法を定義するために java.util.Comparator インタフェースを実装したクラスを作る。以下に例を示す。

```
class ItemComparator implements Comparator<Item> {  // Item の比較方法のクラスの実装例
    @Override
    public int compare(Item a, Item b) {  // a の方が先の順序なら負，同じなら 0，後なら正
        return b.code - a.code;           // 通常とは逆に品番の大きい順を整列順序とする例
    }
}
```

そして，`Arrays.sort(items, new ItemComparetor())` などとして，定義した Comparator クラスのインスタンスを追加の引数として整列・探索のメソッドに渡す。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。

7. 【発展】ArrayList に対しては，java.util.Arrays のメソッドは使えないが，java.util.Collections が提供する Collections.sort および Collections.binarySearch が同様な方法で使える。これらを用いて 4.と同様の処理を行うプログラムを ArrayList<Item>を用いて作成し，動作を確認せよ。

8. 【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.のアルゴリズムの文字比較回数を比べるために，それぞれのプログラムの/* ★ */の if 文の**前に**実行回数を数える処理を加え，それを最後に表示させるように改造せよ。さらに，ある程度長い英文テキストを使って回数を比較してみよ。

```
public class Main {

    // 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 ビットだけ抽出して利用する
    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 key) {

        // 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(key);
        int n = text.length();
        int m = key.length();

        int i = 0;
        while (i <= n - m) {
            // 探索文字列の末尾の文字（m 文字目）から前に 1 文字ずつ照合していく
            int k;
            for (k = m - 1; k >= 0; m--) {
                // テキストから照合位置の文字を取り出し，探索文字列の対応位置の文字と比較する
                int code = text.charAt(i + k);
                if (code != key.charAt(k)) {  /* ★ */
                    // 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
                    i += skipTable[code & BITMASK];
                    break;
                }
            }
            if (k < 0) return i;
        }
        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"));
    }
}
```