

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 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 ItemComparator())` 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、インスタンス生成時に `E` の代わりに任意のクラス名を当てはめることができる。ただし、`int`, `double` などの基本型はクラスではないので、`Integer`, `Double` などのラッパークラスで代用する。以下に、`ArrayList` の使用パターンをいくつか示す。

// 必要な import 文 <code>import java.util.ArrayList;</code>	
// 動的配列の生成と要素の格納 <code>ArrayList<Integer> intArray = new ArrayList<Integer>(); // 整数(ラッパークラス)の動的配列</code> <code>ArrayList<String> strArray = new ArrayList<String>(); // 文字列の動的配列</code> <code>Integer a = new Integer(32); // Integer a = 32; でも自動変換(autoboxing)される</code> <code>intArray.add(a); // 末尾に要素を追加するのは.addを使う(自動的に拡張)</code> <code>intArray.add(64); // 直接intを渡しても自動変換(autoboxing)される</code> <code>strArray.add("abc"); // 文字列(String)はクラスなので問題なく追加される</code>	
// 配列要素へのアクセス <code>int n = intArray.size(); // 要素数を得るには.sizeを使う(通常配列の.lengthと同じ)</code> <code>for (int i = 0; i < n; i++) {</code> <code> int b = intArray.get(i); // i番目の要素を得るには.get(i)とする(auto-unboxing)</code> <code> System.out.println(b);</code> <code>}</code> <code>strArray.set(0, "ABC"); // i番目の要素をxに入れ替えるには.set(i, x)とする</code> <code>for (String str : strArray) // 通常の配列のように拡張 for 文も使える</code> <code> System.out.println(str);</code>	
// 配列要素の探索 <code>int j = intArray.indexOf(64); // 先頭から線形探索でデータを探す</code> <code>int k = intArray.lastIndexOf(32); // 末尾から線形探索でデータを探す</code>	

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、`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うことができる。3.と4.で作成したプログラムを `ArrayList<Item>` を利用するように書き換え、動作を確認せよ。