CSM 61B Spring 2017

Problem Solving with ADTs

Week 6: February 20, 2017

A **list** is an ordered sequence of items: like an array, but without worrying about the length or size.

```
interface List<E> {
    boolean add(E element);
    void add(int index, E element);
    E get(int index);
    int size();
}
```

A **set** is an unordered collection of unique elements.

```
interface Set<E> {
    boolean add(E element);
    boolean contains(Object object);
    int size();
    boolean remove(Object object);
}
```

A **map** is a collection of key-value mappings, like a dictionary in Python. Like a set, the keys in a map are unique.

```
interface Map<K,V> {
    V put(K key, V value);
    V get(K key);
    boolean containsKey(Object key);
    Set<K> keySet();
}
```

1 Practice Problems

1.1 Define a procedure, `hasDuplicates`, which returns `true` when the given `array` contains duplicates.

```
public static boolean hasDuplicates(int[] array) {
    Set<Integer> seen = new HashSet<>();
    for (int value : array) {
        if (seen.contains(value)) {
            return true;
        }
        seen.add(value);
    }
    return false;
}
```

1.2 Define a procedure, `sumUp`, which returns `true` if any two values in the `array` sum up to `n`.

```
public static boolean sumUp(int[] array, int n) {
    Set<Integer> seen = new HashSet<>();
    for (int value : array) {
        if (seen.contains(n - value)) {
            return true;
        }
        seen.add(value);
    }
    return false;
}
```

1.3 An `array` contains all the numbers from 0 to n except for some number, k. Define a procedure, `missingNo`, which returns k given the input `array`.

```
public static int missingNo(int[] array) {
    Set<Integer> seen = new HashSet<>();
    for (int value : array) {
        seen.add(value);
    }
    for (int i = 0; i <= array.length; i++) {
        if (!seen.contains(i)) {
            return i;
        }
    }
    return -1;
}
```

Or, using the sum of the arithmetic series.

```
public static int missingNo(int[] array) {
    int sum = 0;
    for (int value : array) {
        sum += value;
    }
    int expectedSum = (array.length * (array.length + 1)) / 2;
    return expectedSum - sum;
}
```

1.4 Define a procedure, `isPermutation`, which returns `true` if a string `s1` is a permutation of `s2`. For example, `"atc"` and `"tac"` are permutations of `"cat"`.

```
public static boolean isPermutation(String s1, String s2) {
    Map<Character,Integer> characterCounts = new HashMap<>();
    for (char c : s1.toCharArray()) {
        int count = 0;
        if (characterCounts.containsKey(c)) {
            count = characterCounts.get(c);
        }
        characterCounts.put(c, count + 1);
    }
    for (char c : s2.toCharArray()) {
        int count = 0;
        if (characterCounts.containsKey(c)) {
            count = characterCounts.get(c);
        }
        characterCounts.put(c, count - 1);
    }
    for (char c : characterCounts.keySet()) {
        if (characterCounts.get(c) != 0 ) {
            return false;
        }
    }
```

```
    return true;
}
```

1.5 Define `findDuplicatesWithinK`, a procedure which, when given an `int[] array` and an boundary range *k*, returns a `Set` of all duplicates within *k* indices of each other.

```
findDuplicatesWithinK([1, 2, 3, 1, 4, 3], 3) // {1, 3}
findDuplicatesWithinK([1, 2, 3, 1, 4, 3], 2) // {}
```

```
public static Set<Integer> findDuplicatesWithinK(int[] array, int k) {
    Map<Integer,Integer> seen = new HashMap<>();
    Set<Integer> duplicates = new HashSet<>();
    for (int i = 0; i < array.length; i++) {
        if (!seen.containsKey(array[i])) {
            seen.put(array[i], 0);
        }
        if (seen.get(array[i]) > 0) {
            duplicates.add(array[i]);
        }
        seen.put(array[i], seen.get(array[i]) + 1);
        if (i - k >= 0) {
            seen.put(array[i - k], seen.get(array[i - k]) - 1);
        }
    }
    return duplicates;
}
```