

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) {
```

```
}
```

- 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) {
```

```
}
```

- 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) {
```

```
}
```

- 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) {
```

```
}
```

4 Problem Solving with ADTs

- 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) {
```

```
}
```