Two Sum I (LeetCode 1 - Easy)

Problem Description

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have ***exactly*** one solution, and you may not use the *same* element twice.

The input array is unsorted.

Example:

Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1].

Problem Analysis

Use HashMap<Key, Value> → number as key, index as value.

One iteration → O(n)

Solution

```
/**
 * HashMap, use nums as values, indexed as keys
 * @param nums, unsorted
 * @param target
 * @return indices of the two numbers such that they add up to a specific target
 * @complexity O(n)
 */
public static int[] twoSum_1(int[] nums, int target) {
	int[] rv = new int[2];
	HashMap<Integer, Integer> hashmap = new HashMap<Integer, Integer>();
	for (int i = 0; i < nums.length; i++) {
		if (hashmap.containsKey(target - nums[i])) {
			rv[0] = hashmap.get(target - nums[i]);
			rv[1] = i;
			return rv;
		} else {
			if (!hashmap.containsKey(nums[i])) {
				hashmap.put(nums[i], i);
			}
		}
	}
	return rv;
}
```

Two Sum II – Input array is sorted (LeetCode167 - Medium)

Problem Description

Given an array of integers, return **indices** of the two numbers such that they add up to a specific target.

You may assume that each input would have ***exactly*** one solution, and you may not use the *same* element twice.

The input array is sorted.

Example:

Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9, return [0, 1].

Problem Analysis

Two pointers: low and high, collision.

Solution

```
/**
 * Two pointers, low and high, collision.
 * @param nums, sorted
 * @param target
 * @return indices of the two numbers such that they add up to a specific target
 * @complexity O(n)
 */
public static int[] twoSum_2(int[] nums, int target) {
	int[] rv = new int[2];
	int low = 0;
	int high = nums.length - 1;
	while(low < high) {
		if(nums[low] + nums[high] > target) {
			high--;
		}else if(nums[low] + nums[high] < target) {
			low++;
		}else {
			break;
		}
	}
	rv[0] = low + 1;
	rv[1] = high + 1;
	return rv;
}
```

Two Sum III (LintCode443 - Medium)

Problem Description

Given an array of integers, find **indices** of the two numbers such that they add up to a sum that is **larger than** specific target. Return number of such pairs.

Example:

Given nums = [2, 7, 11, 15], target = 23,

Because nums[2] + nums[3] = 11 + 15 = 26 > 23, return 1.

Problem Analysis

Sort first –> O(nlogn)

Two pointers: low and high, collision.

Solution

```
/**
 * Two pointers, collision
 * @param nums an array of numbers
 * @param target
 * @return number of pairs with sum larger than target
 * @complexity O(nlogn)
 */
public static int twoSum_3(int[] nums, int target) {
	int rv = 0;
	Arrays.sort(nums);//O(nlogn)
	int low = 0;
	int high = nums.length-1;
	while(low < high) {
		if(nums[low] + nums[high] <= target) {
			low++;
		}else {
			rv += high - low;
			high--;
		}
	}
	return rv;
}
```

Triangle Count (LintCode382 - Hard)

Problem Description

Given an array of integers, use any three of these numbers as edges of a triangle, return how many triangles these numbers can form.

Example:

Given nums = [3, 4, 6, 7]

We can form three triangles: {3, 4, 6}, {3, 6, 7}, {4, 6, 7}, return 3

Problem Analysis

Sort first → O(nlogn)

Iterate through the array, assume the current integer is one of the edges, for the remaining numbers, use two pointers -> low and high, collision. → O(n^2)

Solution

```
/**
 * @param S: A list of integers.
 * @return Number of triangles these integers can form.
 * @comlexity O(n*n)
 */
public int triangleCount(int S[]) {
    int rv = 0;
    Arrays.sort(S);//nlog(n)
    for(int i = 2; i < S.length; i++) {
            int low = 0;
            int high = i - 1;
            int longestEdge = S[i];
            while(low < high) {
                    if(S[low] + S[high] > longestEdge) {
                            rv += high-low;
                            high--;
                    }else {
                            low++;
                    }
            }
    }
    return rv;
}
```