**[Easy] #001 Two Sum**

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.

**Example:**

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

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

return [0, 1].

**UPDATE (2016/2/13):**The return format had been changed to zero-based indices. Please read the above updated description carefully.

**My Solution 1**

**T: O(2n), S: O(n)**

|  |
| --- |
| **public** **int**[] twoSum(**int**[] nums, **int** target) {  Map<Integer, Integer> map = **new** HashMap<>();  **for** (**int** i = 0; i < nums.length; ++i) {  **if** (!map.containsKey(nums[i])) {  map.put(nums[i], i);  }  }  **for** (**int** i = 0; i < nums.length; ++i) {  **int** leftValue = target - nums[i];  **if** (map.containsKey(leftValue) && map.get(leftValue) != i) {  **return** **new** **int**[] { i, map.get(leftValue) };  }  }  **return** **null**;  } |

**My Solution 2 : One-Pass HashMap**

**T: O(n), S: O(n)**

|  |
| --- |
| **public** **int**[] twoSum (**int**[] nums, **int** target) {  Map<Integer, Integer> map = **new** HashMap<>();  **for** (**int** i = 0; i < nums.length; ++i) {  **if** (!map.containsKey(nums[i])) {  map.put(nums[i], i);  }  **int** leftValue = target - nums[i];  **if** (map.containsKey(leftValue) && map.get(leftValue) != i) {  **return** **new** **int**[] { map.get(leftValue), i };  }  }  **return** **null**;  } |

**Top Solutions**

**T: O(n), S: O(n)**

|  |
| --- |
| **public** **int**[] twoSum(**int**[] numbers, **int** target) {  **int**[] result = **new** **int**[2];  Map<Integer, Integer> map = **new** HashMap<Integer, Integer>();  **for** (**int** i = 0; i < numbers.length; i++) {  **if** (map.containsKey(target - numbers[i])) {  result[1] = i + 1;  result[0] = map.get(target - numbers[i]);  **return** result;  }  map.put(numbers[i], i + 1);  }  **return** result;  } |

**[Medium] #002 Add Two Numbers**

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

**Example:**

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)  
Output:  7 -> 0 -> 8

**My Solution:**

**T: O(max(n,m)), S: O(max(n,m))**

|  |
| --- |
| **public** ListNode addTwoNumbers(ListNode l1, ListNode l2) {  ListNode n1 = l1;  ListNode n2 = l2;  **int** carry = 0;  ListNode result = **new** ListNode(carry);  ListNode node = result;  **while** (n1 != **null** || n2 != **null**) {  **int** val1 = n1 != **null** ? n1.val : 0;  **int** val2 = n2 != **null** ? n2.val : 0;  **int** total = val1 + val2 + node.val;  node.val = total % 10;  carry = total / 10;  **if** (n1 != **null**) n1 = n1.next;  **if** (n2 != **null**) n2 = n2.next;  **if** (n1 != **null** || n2 != **null** || carry != 0) {  node.next = **new** ListNode(carry);  node = node.next;  }  }  **return** result;  } |

**[Medium] #003 Longest Substring Without Repeating Characters**

Given a string, find the length of the longest substring without repeating characters.

**Examples:**

Given "abcabcbb", the answer is "abc", which the length is 3.

Given "bbbbb", the answer is "b", with the length of 1.

Given "pwwkew", the answer is "wke", with the length of 3. Note that the answer must be a substring, "pwke" is a subsequence and not a substring.

**Solution 1**

**T: O(n), S: O(n)**

|  |
| --- |
| **public** **int** lengthOfLongestSubstring(String s) {  **char**[] cArray = s.toCharArray();  **int** maxLength = 0;  **int** startIndex = 0;  HashMap<Character, Integer> map = **new** HashMap<>();  **for** (**int** i = 0; i < cArray.length; ++i) {  **if** (map.containsKey(cArray[i])) {  **int** length = i - startIndex;  **if** (length > maxLength) {  maxLength = length;  }  **int** index = map.get(cArray[i]) + 1;  **if** (index > startIndex) {  startIndex = index;  }  }  map.put(cArray[i], i);  }  **int** length = cArray.length - startIndex;  **if** (length > maxLength) maxLength = length;  **return** maxLength;  } |

**Solution 2: Sliding Windows**

**T: O(2n), S: O(min(n,m))**

|  |
| --- |
| **public** **int** lengthOfLongestSubstring(String s) {  **int** length = s.length();  **int** maxLength = 0;  **int** i = 0;  **int** j = 0;  Set<Character> set = **new** HashSet<>();  **while** (i < length && j < length) {  **char** c = s.charAt(j);  **if** (!set.contains(c)) {  **Time complexity: O(2n) = O(n)**  In the worst case each character will be visited twice by i and j.  **Space complexity: O(min(m,n))**  Same as the previous approach. We need O(k) space for the sliding window, where k is the size of the Set. The size of the Set is upper bounded by the size of the string n  set.add(c);  ++j;  maxLength = Math.*max*(maxLength, j - i);  } **else** {  set.remove(s.charAt(i));  ++i;  }  }  **return** maxLength;  } |

**[Medium] #003 Longest Substring Without Repeating Characters**

Given a string, find the length of the longest substring without repeating characters.

**Solution 3: Sliding Windows Optimized**

**T: O(n), S: O(min(n,m))**

|  |
| --- |
| **public** **int** lengthOfLongestSubstring(String s) {  **int** length = s.length();  **int** maxLength = 0;  Map<Character, Integer> map = **new** HashMap<>();  **int** i = 0;  **int** j = 0;  **while** (i < length && j < length) {  **char** c = s.charAt(j);  **if** (map.containsKey(c)) {  i = Math.*max*(map.get(c) + 1, i);  **Time complexity: O(n)**  Index j will iterate n times.  **Space complexity (HashMap): O(min(m,n))** Same as the previous approach.  **Space complexity (Table): O(m)**  m is the size of the charset.  }  maxLength = Math.*max*(maxLength, j - i + 1);  map.put(c, j);  ++j;  }  **return** maxLength;  } |

**Solution 4:**

|  |
| --- |
| /\*  \* Java (Assuming ASCII 128)  \*  \* The previous implements all have no assumption on the charset of the string s.  \* If we know that the charset is rather small, we can replace the Map with an integer array as direct access table.  \*  \* Commonly used tables are:  \* int[26] for Letters 'a' - 'z' or 'A' - 'Z'  \* int[128] for ASCII  \* int[256] for Extended ASCII  \*/  **public** **int** lengthOfLongestSubstring(String s) {  **int** length = s.length();  **int**[] index = **new** **int**[128]; // default value is 0  **int** maxLength = 0;  **int** i = 0;  **int** j = 0;  **while** (i < length && j < length) {  **char** c = s.charAt(j);  i = Math.*max*(index[c], i);  maxLength = Math.*max*(maxLength, j - i + 1);  index[c] = ++j; // record the next index of current char  }  **return** maxLength;  } |

**Solution 5:**

**T: O(n), S:O(?)**

|  |
| --- |
| **public** **int** lengthOfLongestSubstring(String s) {  **if** (s.length() == 0) **return** 0;  HashMap<Character, Integer> map = **new** HashMap<>();  **int** max = 0;  **for** (**int** i = 0, j = 0; i < s.length(); ++i) {  **if** (map.containsKey(s.charAt(i))) {  j = Math.*max*(j, map.get(s.charAt(i)) + 1);  }  map.put(s.charAt(i), i);  the basic idea is, keep a hashmap which stores the characters in string as keys and their positions as values, and keep two pointers which define the max substring. move the right pointer to scan through the string , and meanwhile update the hashmap. If the character is already in the hashmap, then move the left pointer to the right of the same character last found. Note that the two pointers can only move forward.  max = Math.*max*(max, i – j + 1);  }  **return** max;  } |

**Utility**

**Class ListNode**

|  |
| --- |
| **public** **class** ListNode {  **int** val;  ListNode next;  ListNode (**int** x) {  **this**(x, **null**);  }  ListNode (**int** x, ListNode n) {  val = x;  next = n;  }  } |

**Class TreeNode**

|  |
| --- |
| **public** **class** TreeNode {  **int** val;  TreeNode left;  TreeNode right;  TreeNode(**int** x) { val = x; }  } |

**Class TreeLinkNode**

|  |
| --- |
| **public** **static** **class** TreeLinkNode {  **int** val;  TreeLinkNode left;  TreeLinkNode right;  TreeLinkNode next;  TreeLinkNode(**int** x) { val = x; }  } |