**[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;  } |

**[Medium] #006 ZigZag Conversion**

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)



And then read line by line: "PAHNAPLSIIGYIR"

Write the code that will take a string and make this conversion given a number of rows:



convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".

**My Solution:**

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

|  |
| --- |
| **public** String convert(String s, **int** numRows) {  **if** (numRows == 1) **return** s;  StringBuilder sb = **new** StringBuilder();  **int** delta = numRows + numRows - 2;  **int** r = 0;  **while** (r < numRows) {  **int** l = r;  **boolean** flag = **false**;  **while** (l < s.length()) {  sb.append(s.charAt(l));  **if** (r % (numRows - 1) == 0) { // the first and last lines  l += delta;  } **else** **if** (flag) {  l += r \* 2;  flag = **false**;  } **else** {  l += delta - r \* 2;  flag = **true**;  }  }  ++r;  }  **return** sb.toString();  } |

**Solution:**

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

|  |
| --- |
| **public** String convert(String s, **int** nRows) {  **char**[] c = s.toCharArray();  **int** len = c.length;  StringBuffer[] sb = **new** StringBuffer[nRows];  **for** (**int** i = 0; i < sb.length; i++) {  sb[i] = **new** StringBuffer();  }  **int** i = 0;  **while** (i < len) {  **for** (**int** idx = 0; idx < nRows && i < len; idx++) // vertically down  sb[idx].append(c[i++]);  **for** (**int** idx = nRows-2; idx >= 1 && i < len; idx--) // obliquely up  sb[idx].append(c[i++]);  }  **for** (**int** idx = 1; idx < sb.length; idx++)  sb[0].append(sb[idx]);  **return** sb[0].toString();  } |

**[Easy] #007 Reverse Integer**

Reverse digits of an integer.

**Example1:** x = 123, return 321  
**Example2:** x = -123, return -321

**Have you thought about this?**

Here are some good questions to ask before coding. Bonus points for you if you have already thought through this!

If the integer's last digit is 0, what should the output be? ie, cases such as 10, 100.

Did you notice that the reversed integer might overflow? Assume the input is a 32-bit integer, then the reverse of 1000000003 overflows. How should you handle such cases?

For the purpose of this problem, assume that your function returns 0 when the reversed integer overflows.

**Note:**  
The input is assumed to be a 32-bit signed integer. Your function should return 0 when the reversed integer overflows.

**My Solution:**

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

|  |
| --- |
| **public** **int** reverse(**int** x) {  **boolean** isNegative = **false**;  **int** num = x;  **if** (num < 0) {  num \*= -1;  isNegative = **true**;  }  **long** result = 0; // define result as long type to check if overflow  **while** (num > 0) {  result \*= 10;  **int** left = num % 10;  result += left;  **if** (result > Integer.***MAX\_VALUE***) **return** 0; // if overflow  num = num / 10;  }  **if** (isNegative) {  result \*= -1;  }  **return** (**int**) result;  } |

**Solution:**

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

|  |
| --- |
| **public** **int** reverse(**int** x) {  **int** result = 0;  **while** (x != 0) {  **int** tail = x % 10;  **int** newResult = result \* 10 + tail;  // If overflow exists, the new result will not equal previous one  **if** ((newResult - tail) / 10 != result) **return** 0;  result = newResult;  x = x / 10;  }  **return** result;  } |

**[Medium] #008 String to Integer (atoi)**

Implement atoi to convert a string to an integer.

**Hint:** Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.

**Notes:** It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.

**Requirements for atoi:**

The function first discards as many whitespace characters as necessary until the first non-whitespace character is found. Then, starting from this character, takes an optional initial plus or minus sign followed by as many numerical digits as possible, and interprets them as a numerical value.

The string can contain additional characters after those that form the integral number, which are ignored and have no effect on the behavior of this function.

If the first sequence of non-whitespace characters in str is not a valid integral number, or if no such sequence exists because either str is empty or it contains only whitespace characters, no conversion is performed.

If no valid conversion could be performed, a zero value is returned. If the correct value is out of the range of representable values, INT\_MAX (2147483647) or INT\_MIN (-2147483648) is returned.

**My Solution:**

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

|  |
| --- |
| **public** **int** myAtoi(String str) {  **if** (str == **null**) **return** 0;  str = str.trim();  **if** (str.length() == 0) **return** 0;  **long** result = 0;  **boolean** isNegative = **false**;  **int** i = 0;  **char** c = str.charAt(i);  **if** (c == '-' || c == '+') {  isNegative = (c == '-');  i++;  }  **while** (i < str.length()) {  c = str.charAt(i++);  **if** (c < '0' || c > '9') **break**;  result = result \* 10 + (c - '0');  **if** (!isNegative && result >= Integer.***MAX\_VALUE***) **return** Integer.***MAX\_VALUE***;  **if** (isNegative && (-1) \* result <= Integer.***MIN\_VALUE***) **return** Integer.***MIN\_VALUE***;  }  **if** (isNegative) result \*= -1;  **return** (**int**)result;  } |

**Solution:**

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

|  |
| --- |
| **public** **int** myAtoi(String str) {  **int** index = 0, sign = 1, total = 0;  **if**(str.length() == 0) **return** 0; //1. Empty string  **while**(str.charAt(index) == ' ' && index < str.length()) index++; //2. Remove Spaces  **if**(str.charAt(index) == '+' || str.charAt(index) == '-') { //3. Handle signs  sign = str.charAt(index) == '+' ? 1 : -1;  index++;  }  **while**(index < str.length()) { //4. Convert number and avoid overflow  **int** digit = str.charAt(index) - '0';  **if**(digit < 0 || digit > 9) **break**;  //check if total will be overflow after 10 times and add digit  **if**(Integer.***MAX\_VALUE*** / 10 < total || Integer.***MAX\_VALUE*** / 10 == total && Integer.***MAX\_VALUE*** % 10 < digit)  **return** sign == 1 ? Integer.***MAX\_VALUE*** : Integer.***MIN\_VALUE***;  total = 10 \* total + digit;  index ++;  }  **return** total \* sign;  } |

**[Easy] #009 Palindrome Number**

Determine whether an integer is a palindrome. Do this without extra space.

**Some hints:**

Could negative integers be palindromes? (ie, -1)

If you are thinking of converting the integer to string, note the restriction of using extra space.

You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case?

There is a more generic way of solving this problem.

**My Solution:**

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

|  |
| --- |
| **public** **boolean** isPalindrome(**int** x) {  **if** (x < 0) **return** **false**;  **if** (x == 0) **return** **true**;  **int** length = isPalindrome\_getLengthOfInt(x);  **int** factor = isPalindrome\_getFactor(length);  **while** (x > 9) { // 12321 -> 232 -> 3  // 1221 -> 22 -> 0  **int** a = x / factor;  **int** b = x % 10;  **if** (a != b) **return** **false**;  x = (x - a \* factor - b) / 10;  factor = factor / 100;  }  **if** (x == 0) **return** **true**;  **if** (factor != 1 && factor != 0) **return** **false**;  **return** **true**;  }  **private** **int** isPalindrome\_getLengthOfInt(**int** x) {  **if** (x == 0) **return** 1;  **int** length = 0;  **int** num = x;  **while** (num > 0) {  length++;  num = num / 10;  }  **return** length;  }  **private** **int** isPalindrome\_getFactor(**int** length) {  **int** factor = 1;  **for** (**int** i = 0; i < length - 1; ++i) {  factor \*= 10;  }  **return** factor;  } |

**Solution:**

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

|  |
| --- |
| **public** **boolean** isPalindrome(**int** x) {  **if** (x < 0 || (x != 0 && x % 10 == 0)) **return** **false**;  **int** rev = 0;  **while** (x > rev) {  rev = rev \* 10 + x % 10;  x = x / 10;  }  **return** (x == rev || x == rev / 10);  } |

**[Medium] #012 Integer to Roman**

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

**My Solution:**

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

|  |
| --- |
| **public** String intToRoman(**int** num) {  Map<Integer, String> romanMap = **new** HashMap<Integer, String>();  romanMap.put(1000, "M");  romanMap.put(500, "D");  romanMap.put(100, "C");  romanMap.put(50, "L");  romanMap.put(10, "X");  romanMap.put(5, "V");  romanMap.put(1, "I");  StringBuilder sb = **new** StringBuilder();  **int** n = num;  **int** factor = 1000;  **while** (n > 0) {  **int** left = n / factor;  **if** (left > 0) {  sb.append(intToRoman\_digitToRoman(left, factor, romanMap));  }  n = n - left \* factor;  factor = factor / 10;  }  **return** sb.toString();  }  **private** String intToRoman\_digitToRoman(**int** digit, **int** factor, Map<Integer, String> map) {  StringBuilder sb = **new** StringBuilder();  **if** (digit == 9) {  sb.append(map.get(factor));  sb.append(map.get(factor \* 10));  **return** sb.toString();  }  **else** **if** (digit == 5) {  **return** map.get(digit \* factor);  }  **else** **if** (digit == 4) {  sb.append(map.get(factor));  sb.append(map.get(5 \* factor));  **return** sb.toString();  }  **else** **if** (digit < 4) { // 1,2,3  String sig = map.get(factor);  **for** (**int** i = 0; i < digit; ++i) {  sb.append(sig);  }  **return** sb.toString();  }  **else** { // digit 6,7,8  sb.append(map.get(5 \* factor));  **for** (**int** i = 5; i < digit; ++i) {  sb.append(map.get(factor));  }  **return** sb.toString();  }  } |

**Solution 1:**

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

|  |
| --- |
| **public** String intToRoman(**int** num) {  String M[] = {"", "M", "MM", "MMM"};  String C[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};  String X[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};  String I[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};  **return** M[num / 1000] + C[(num % 1000) / 100] + X[(num % 100) / 10] + I[num % 10];  } |

**[Medium] #012 Integer to Roman**

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

**Solution 2:**

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

|  |
| --- |
| **public** String intToRoman(**int** num) {  **int**[] values = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1 };  String[] strs = {"M","CM", "D", "CD","C","XC","L","XL","X","IX","V","IV","I"};  StringBuilder sb = **new** StringBuilder();  **for**(**int** i=0;i<values.length;i++) {  **while**(num >= values[i]) {  num -= values[i];  sb.append(strs[i]);  }  }  **return** sb.toString();  } |

**[Easy] #013 Roman to Integer**

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

**My Solution:**

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

|  |
| --- |
| **public** **int** romanToInt(String s) {  **if** (s == **null** || s.length() == 0) **return** 0;  Map<Character, Integer> romanMap = **new** HashMap<Character, Integer>();  romanMap.put('M', 1000);  romanMap.put('D', 500);  romanMap.put('C', 100);  romanMap.put('L', 50);  romanMap.put('X', 10);  romanMap.put('V', 5);  romanMap.put('I', 1);  **int** result = 0;  **int** prev = -1;  **for** (**int** i = s.length() - 1; i >= 0; --i) {  **int** val = romanMap.get(s.charAt(i));  **if** (val < prev) {  val \*= -1;  }  result += val;  prev = val;  }  **return** result;  } |

**[Easy] #014 Longest Common Prefix**

Write a function to find the longest common prefix string amongst an array of strings.

**My Solution 1:**

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

|  |
| --- |
| **public** String longestCommonPrefix(String[] strs) {  **if** (strs.length == 0) **return** "";  **int** minLength = Integer.***MAX\_VALUE***;  **int** index = 0;  **for** (**int** i = 0; i < strs.length; ++i) {  **if** (strs[i].length() < minLength) {  minLength = strs[i].length();  index = i;  }  }  String prefix = strs[index];  **while** (prefix.length() != 0) {  **int** j = 0;  **for** (;j < strs.length; ++j) {  **if** (!strs[j].startsWith(prefix)) {  **break**;  }  }  **if** (j >= strs.length) **return** prefix;  prefix = prefix.substring(0, prefix.length() - 1);  }  **return** prefix;  } |

**My Solution 2:**

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

|  |
| --- |
| **public** String longestCommonPrefix(String[] strs) {  **if** (strs.length == 0) **return** "";  **int** minLength = Integer.***MAX\_VALUE***;  **int** index = 0;  **for** (**int** i = 0; i < strs.length; ++i) {  **if** (strs[i].length() < minLength) {  minLength = strs[i].length();  index = i;  }  }  String prefix = strs[index];  **int** i = 0; // index of prefix  **while** (i < prefix.length()) {  **char** c1 = prefix.charAt(i);  **int** j = 0; // index of String array  **while** (j < strs.length) {  **char** c2 = strs[j].charAt(i);  **if** (c1 != c2) **break**;  ++j;  }  **if** (j >= strs.length) ++i;  **else** **break**;  }  **return** prefix.substring(0, i);  } |

**[Easy] #014 Longest Common Prefix**

Write a function to find the longest common prefix string amongst an array of strings.

**Solution 1: (2ms)**

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

|  |
| --- |
| **public** String longestCommonPrefix(String[] strs) {  StringBuilder result = **new** StringBuilder();  **if** (strs!= **null** && strs.length > 0) {  Arrays.sort(strs); // Sort the array first  **char** [] a = strs[0].toCharArray(); //  then you can simply compare the first and last elements in the sorted array  **char** [] b = strs[strs.length-1].toCharArray();  **for** (**int** i = 0; i < a.length; i ++) {  **if** (b.length > i && b[i] == a[i]) {  result.append(b[i]);  }  **else** {  **return** result.toString();  }  }  }  **return** result.toString();  } |

**Solution 2: (13 lines)**

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

|  |
| --- |
| **public** String longestCommonPrefix(String[] strs) {  **if**(strs == **null** || strs.length == 0) **return** "";  String pre = strs[0];  **int** i = 1;  **while**(i < strs.length) {  **while**(strs[i].indexOf(pre) != 0)  pre = pre.substring(0, pre.length()-1);  i++;  }  **return** pre;  } |

**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; }  } |