**Coding Interview in Java**

**Program Creek**

May 1st, 2016

1 of 246 1 of 288

**Contents**

**1 Rotate Array in Java 13 2 Evaluate Reverse Polish Notation 17 3 Isomorphic Strings 21 4 Word Ladder 23 5 Word Ladder II 25 6 Median of Two Sorted Arrays 29 7 Kth Largest Element in an Array 31 8 Wildcard Matching 33 9 Regular Expression Matching in Java 35 10 Merge Intervals 39 11 Insert Interval 41 12 Two Sum 43 13 Two Sum II Input array is sorted 45 14 Two Sum III Data structure design 47 15 3Sum 49 16 4Sum 53 17 3Sum Closest 55 18 String to Integer (atoi) 57 19 Merge Sorted Array 59 20 Valid Parentheses 61**

2 | 531

2 of 246 2 of 288

Contents

**21 Longest Valid Parentheses 63 22 Implement strStr() 65 23 Minimum Size Subarray Sum 69 24 Search Insert Position 73 25 Longest Consecutive Sequence 75 26 Valid Palindrome 77 27 ZigZag Conversion 81 28 Add Binary 83 29 Length of Last Word 85 30 Triangle 87 31 Contains Duplicate 89 32 Contains Duplicate II 91 33 Contains Duplicate III 93 34 Remove Duplicates from Sorted Array 95 35 Remove Duplicates from Sorted Array II 99 36 Longest Substring Without Repeating Characters 103 37 Longest Substring Which Contains 2 Unique Characters 105 38 Substring with Concatenation of All Words 109 39 Minimum Window Substring 111 40 Reverse Words in a String 113 41 Find Minimum in Rotated Sorted Array 115 42 Find Minimum in Rotated Sorted Array II 117 43 Search in Rotated Sorted Array 119 44 Search in Rotated Sorted Array II 121 45 Min Stack 123**

Program Creek 3 | 531

3 of 246 3 of 288

Contents

**46 Majority Element 125 47 Majority Element II 127 48 Remove Element 129 49 Largest Rectangle in Histogram 131 50 Longest Common Prefix 133 51 Largest Number 135 52 Simplify Path 137 53 Compare Version Numbers 139 54 Gas Station 141 55 Pascal’s Triangle 143 56 Pascal’s Triangle II 145 57 Container With Most Water 147 58 Candy 149 59 Trapping Rain Water 151 60 Count and Say 153 61 Search for a Range 155 62 Basic Calculator 157 63 Group Anagrams 159 64 Shortest Palindrome 161 65 Rectangle Area 163 66 Summary Ranges 165 67 Increasing Triplet Subsequence 167 68 Get Target Number Using Number List and Arithmetic Operations 169 69 Reverse Vowels of a String 171 70 Flip Game 173**

4 | 531 Program Creek

4 of 246 4 of 288

Contents

**71 Flip Game II 175 72 Move Zeroes 177 73 Valid Anagram 179 74 Group Shifted Strings 181 75 Top K Frequent Elements 183 76 Find Peak Element 185 77 Word Pattern 187 78 Set Matrix Zeroes 189 79 Spiral Matrix 193 80 Spiral Matrix II 197 81 Search a 2D Matrix 199 82 Search a 2D Matrix II 201 83 Rotate Image 205 84 Valid Sudoku 207 85 Minimum Path Sum 209 86 Unique Paths 211 87 Unique Paths II 213 88 Number of Islands 215 89 Number of Islands II 217 90 Surrounded Regions 219 91 Maximal Rectangle 223 92 Maximal Square 225 93 Word Search 227 94 Word Search II 229 95 Integer Break 233**

Program Creek 5 | 531

5 of 246 5 of 288

Contents

**96 Range Sum Query 2D Immutable 235 97 Longest Increasing Path in a Matrix 237 98 Implement a Stack Using an Array in Java 239 99 Add Two Numbers 243 100 Reorder List 245 101 Linked List Cycle 251 102 Copy List with Random Pointer 253 103 Merge Two Sorted Lists 257 104 Odd Even Linked List 259 105 Remove Duplicates from Sorted List 261 106 Remove Duplicates from Sorted List II 263 107 Partition List 265 108 LRU Cache 267 109 Intersection of Two Linked Lists 271 110 Remove Linked List Elements 273 111 Swap Nodes in Pairs 275 112 Reverse Linked List 277 113 Reverse Linked List II 279 114 Remove Nth Node From End of List 281 115 Implement Stack using Queues 283 116 Implement Queue using Stacks 285 117 Palindrome Linked List 287 118 Implement a Queue using an Array in Java 289 119 Delete Node in a Linked List 291 120 Moving Average from Data Stream 293**

6 | 531 Program Creek

6 of 246 6 of 288

**1 Rotate Array in Java**

You may have been using Java for a while. Do you think a simple Java array question can be a challenge? Let’s use the following problem to test.

Problem: Rotate an array of n elements to the right by k steps. For example, with n = 7 and k = 3, the array [1,2,3,4,5,6,7] is rotated to [5,6,7,1,2,3,4]. How many different ways do you know to solve this problem?

**1.1 Solution 1 - Intermediate Array**

In a straightforward way, we can create a new array and then copy elements to the new array. Then change the original array by using System.arraycopy().

public void rotate(int[] nums, int k) {

if(k > nums.length)

k=k%nums.length;

int[] result = new int[nums.length];

for(int i=0; i < k; i++){

result[i] = nums[nums.length-k+i];

}

int j=0;

for(int i=k; i<nums.length; i++){

result[i] = nums[j];

j++;

}

System.arraycopy( result, 0, nums, 0, nums.length );

}

Space is O(n) and time is O(n). You can check out the difference between Sys tem.arraycopy() and Arrays.copyOf().

**1.2 Solution 2 - Bubble Rotate**

Can we do this in O(1) space?

This solution is like a bubble sort.

public static void rotate(int[] arr, int order) {

if (arr == null || order < 0) {

13 | 531

7 of 246 7 of 288

1 Rotate Array in Java

throw new IllegalArgumentException("Illegal argument!"); }

for (int i = 0; i < order; i++) {

for (int j = arr.length - 1; j > 0; j--) {

int temp = arr[j];

arr[j] = arr[j - 1];

arr[j - 1] = temp;

}

}

}

However, the time is O(n\*k).

Here is an example (length=7, order=3):

i=0

0 1 2 3 4 5 6

0 1 2 3 4 6 5

...

6 0 1 2 3 4 5

i=1

6 0 1 2 3 5 4

...

5 6 0 1 2 3 4

i=2

5 6 0 1 2 4 3

...

4 5 6 0 1 2 3

**1.3 Solution 3 - Reversal**

Can we do this in O(1) space and in O(n) time? The following solution does. Assuming we are given 1,2,3,4,5,6 and order 2. The basic idea is:

1. Divide the array two parts: 1,2,3,4 and 5, 6

2. Reverse first part: 4,3,2,1,5,6

3. Reverse second part: 4,3,2,1,6,5

4. Reverse the whole array: 5,6,1,2,3,4

public static void rotate(int[] arr, int order) {

if (arr == null || arr.length==0 || order < 0) {

throw new IllegalArgumentException("Illegal argument!"); }

if(order > arr.length){

order = order %arr.length;

}

14 | 531 Program Creek

8 of 246 8 of 288

1 Rotate Array in Java

//length of first part

int a = arr.length - order;

reverse(arr, 0, a-1);

reverse(arr, a, arr.length-1);

reverse(arr, 0, arr.length-1);

}

public static void reverse(int[] arr, int left, int right){

if(arr == null || arr.length == 1)

return;

while(left < right){

int temp = arr[left];

arr[left] = arr[right];

arr[right] = temp;

left++;

right--;

}

}

Program Creek 15 | 531

9 of 246 9 of 288

**2 Evaluate Reverse Polish Notation**

Evaluate the value of an arithmetic expression in Reverse Polish Notation. Valid op erators are +, -, \*, /. Each operand may be an integer or another expression. For example:

["2", "1", "+", "3","\*"] -> ((2 + 1) \* 3) -> 9

["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6

**2.1 Naive Approach**

This problem can be solved by using a stack. We can loop through each element in the given array. When it is a number, push it to the stack. When it is an operator, pop two numbers from the stack, do the calculation, and push back the result.

The following is the code. However, this code contains compilation errors in leet code. Why?

public class Test {

public static void main(String[] args) throws IOException {

String[] tokens = new String[] { "2", "1", "+", "3","\*" };

System.out.println(evalRPN(tokens));

}

17 | 531

10 of 246 11 of 288

2 Evaluate Reverse Polish Notation

public static int evalRPN(String[] tokens) {

int returnValue = 0;

String operators = "+-\*/";

Stack<String> stack = new Stack<String>();

for (String t : tokens) {

if (!operators.contains(t)) { //push to stack if it is a number stack.push(t);

} else {//pop numbers from stack if it is an operator

int a = Integer.valueOf(stack.pop());

int b = Integer.valueOf(stack.pop());

switch (t) {

case "+":

stack.push(String.valueOf(a + b));

break;

case "-":

stack.push(String.valueOf(b - a));

break;

case "\*":

stack.push(String.valueOf(a \* b));

break;

case "/":

stack.push(String.valueOf(b / a));

break;

}

}

}

returnValue = Integer.valueOf(stack.pop());

return returnValue;

}

}

The problem is that switch string statement is only available from JDK 1.7. Leetcode apparently use a JDK version below 1.7.

**2.2 Accepted Solution**

If you want to use switch statement, you can convert the above by using the following code which use the index of a string "+-\*/".

public class Solution {

public int evalRPN(String[] tokens) {

int returnValue = 0;

18 | 531 Program Creek

11 of 246 12 of 288

2 Evaluate Reverse Polish Notation

String operators = "+-\*/";

Stack<String> stack = new Stack<String>();

for(String t : tokens){

if(!operators.contains(t)){

stack.push(t);

}else{

int a = Integer.valueOf(stack.pop());

int b = Integer.valueOf(stack.pop());

int index = operators.indexOf(t);

switch(index){

case 0:

stack.push(String.valueOf(a+b));

break;

case 1:

stack.push(String.valueOf(b-a));

break;

case 2:

stack.push(String.valueOf(a\*b));

break;

case 3:

stack.push(String.valueOf(b/a));

break;

}

}

}

returnValue = Integer.valueOf(stack.pop());

return returnValue;

}

}

Program Creek 19 | 531

12 of 246 13 of 288

**3 Isomorphic Strings**

Given two strings s and t, determine if they are isomorphic. Two strings are isomor phic if the characters in s can be replaced to get t.

For example,"egg" and "add" are isomorphic, "foo" and "bar" are not.

**3.1 Analysis**

We need to define a method which accepts a map & a value, and returns the value’s key in the map.

**3.2 Java Solution**

public boolean isIsomorphic(String s, String t) {

if(s==null || t==null)

return false;

if(s.length() != t.length())

return false;

if(s.length()==0 && t.length()==0)

return true;

HashMap<Character, Character> map = new HashMap<Character,Character>(); for(int i=0; i<s.length(); i++){

char c1 = s.charAt(i);

char c2 = t.charAt(i);

Character c = getKey(map, c2);

if(c != null && c!=c1){

return false;

}else if(map.containsKey(c1)){

if(c2 != map.get(c1))

return false;

}else{

map.put(c1,c2);

}

}

return true;

}

21 | 531

13 of 246 15 of 288

3 Isomorphic Strings

// a method for getting key of a target value

public Character getKey(HashMap<Character,Character> map, Character target){ for (Map.Entry<Character,Character> entry : map.entrySet()) { if (entry.getValue().equals(target)) {

return entry.getKey();

}

}

return null;

}

22 | 531 Program Creek

14 of 246 16 of 288

**4 Word Ladder**

Given two words (start and end), and a dictionary, find the length of shortest transfor mation sequence from start to end, such that only one letter can be changed at a time and each intermediate word must exist in the dictionary. For example, given:

start = "hit"

end = "cog"

dict = ["hot","dot","dog","lot","log"]

One shortest transformation is "hit" ->"hot" ->"dot" ->"dog" ->"cog", the program should return its length 5.

**4.1 Analysis**

UPDATED on 06/07/2015.

So we quickly realize that this is a search problem, and breath-first search guarantees the optimal solution.

**4.2 Java Solution**

class WordNode{

String word;

int numSteps;

public WordNode(String word, int numSteps){

this.word = word;

this.numSteps = numSteps;

}

}

23 | 531

15 of 246 17 of 288

4 Word Ladder

public class Solution {

public int ladderLength(String beginWord, String endWord, Set<String> wordDict) {

LinkedList<WordNode> queue = new LinkedList<WordNode>(); queue.add(new WordNode(beginWord, 1));

wordDict.add(endWord);

while(!queue.isEmpty()){

WordNode top = queue.remove();

String word = top.word;

if(word.equals(endWord)){

return top.numSteps;

}

char[] arr = word.toCharArray();

for(int i=0; i<arr.length; i++){

for(char c=’a’; c<=’z’; c++){

char temp = arr[i];

if(arr[i]!=c){

arr[i]=c;

}

String newWord = new String(arr);

if(wordDict.contains(newWord)){

queue.add(new WordNode(newWord, top.numSteps+1));

wordDict.remove(newWord);

}

arr[i]=temp;

}

}

}

return 0;

}

}

24 | 531 Program Creek

16 of 246 18 of 288

**5 Word Ladder II**

Given two words (start and end), and a dictionary, find all shortest transformation sequence(s) from start to end, such that: 1) Only one letter can be changed at a time, 2) Each intermediate word must exist in the dictionary.

For example, given: start = "hit", end = "cog", and dict = ["hot","dot","dog","lot","log"], return:

[

["hit","hot","dot","dog","cog"],

["hit","hot","lot","log","cog"]

]

**5.1 Analysis**

This is an extension of Word Ladder.

The idea is the same. To track the actual ladder, we need to add a pointer that points to the previous node in the WordNode class.

In addition, the used word can not directly removed from the dictionary. The used word is only removed when steps change.

**5.2 Java Solution**

class WordNode{

String word;

int numSteps;

WordNode pre;

public WordNode(String word, int numSteps, WordNode pre){

this.word = word;

this.numSteps = numSteps;

this.pre = pre;

}

}

public class Solution {

public List<List<String>> findLadders(String start, String end, Set<String> dict) {

List<List<String>> result = new ArrayList<List<String>>();

25 | 531

17 of 246 19 of 288

5 Word Ladder II

LinkedList<WordNode> queue = new LinkedList<WordNode>(); queue.add(new WordNode(start, 1, null));

dict.add(end);

int minStep = 0;

HashSet<String> visited = new HashSet<String>(); HashSet<String> unvisited = new HashSet<String>(); unvisited.addAll(dict);

int preNumSteps = 0;

while(!queue.isEmpty()){

WordNode top = queue.remove();

String word = top.word;

int currNumSteps = top.numSteps;

if(word.equals(end)){

if(minStep == 0){

minStep = top.numSteps;

}

if(top.numSteps == minStep && minStep !=0){

//nothing

ArrayList<String> t = new ArrayList<String>();

t.add(top.word);

while(top.pre !=null){

t.add(0, top.pre.word);

top = top.pre;

}

result.add(t);

continue;

}

}

if(preNumSteps < currNumSteps){

unvisited.removeAll(visited);

}

preNumSteps = currNumSteps;

char[] arr = word.toCharArray();

for(int i=0; i<arr.length; i++){

for(char c=’a’; c<=’z’; c++){

char temp = arr[i];

if(arr[i]!=c){

arr[i]=c;

}

26 | 531 Program Creek

18 of 246 20 of 288

5 Word Ladder II

String newWord = new String(arr);

if(unvisited.contains(newWord)){

queue.add(new WordNode(newWord, top.numSteps+1, top));

visited.add(newWord);

}

arr[i]=temp;

}

}

}

return result;

}

}

Program Creek 27 | 531

19 of 246 21 of 288

**6 Median of Two Sorted Arrays**

There are two sorted arrays A and B of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).

**6.1 Java Solution 1**

If we see log(n), we should think about using binary something.

This problem can be converted to the problem of finding kth element, k is (A’s length + B’ Length)/2.

If any of the two arrays is empty, then the kth element is the non-empty array’s kth element. If k == 0, the kth element is the first element of A or B.

For normal cases(all other cases), we need to move the pointer at the pace of half of an array length to get log(n) time.

public static double findMedianSortedArrays(int A[], int B[]) { int m = A.length;

int n = B.length;

if ((m + n) % 2 != 0) // odd

return (double) findKth(A, B, (m + n) / 2, 0, m - 1, 0, n - 1); else { // even

return (findKth(A, B, (m + n) / 2, 0, m - 1, 0, n - 1)

+ findKth(A, B, (m + n) / 2 - 1, 0, m - 1, 0, n - 1)) \* 0.5; }

}

public static int findKth(int A[], int B[], int k,

int aStart, int aEnd, int bStart, int bEnd) {

int aLen = aEnd - aStart + 1;

int bLen = bEnd - bStart + 1;

// Handle special cases

if (aLen == 0)

return B[bStart + k];

if (bLen == 0)

return A[aStart + k];

if (k == 0)

return A[aStart] < B[bStart] ? A[aStart] : B[bStart];

int aMid = aLen \* k / (aLen + bLen); // a’s middle count

int bMid = k - aMid - 1; // b’s middle count

29 | 531

20 of 246 23 of 288

6 Median of Two Sorted Arrays

// make aMid and bMid to be array index

aMid = aMid + aStart;

bMid = bMid + bStart;

if (A[aMid] > B[bMid]) {

k = k - (bMid - bStart + 1);

aEnd = aMid;

bStart = bMid + 1;

} else {

k = k - (aMid - aStart + 1);

bEnd = bMid;

aStart = aMid + 1;

}

return findKth(A, B, k, aStart, aEnd, bStart, bEnd);

}

**6.2 Java Solution 2**

Solution 1 is a general solution to find the kth element. We can also come up with a simpler solution which only finds the median of two sorted arrays for this particular problem. Thanks to Gunner86. The description of the algorithm is awesome!

1) Calculate the medians m1 and m2 of the input arrays ar1[] and ar2[] respectively.

2) If m1 and m2 both are equal then we are done, and return m1 (or m2) 3) If m1 is greater than m2, then median is present in one of the below two subarrays.

a) From first element of ar1 to m1 (ar1[0...|\_n/2\_|])

b) From m2 to last element of ar2 (ar2[|\_n/2\_|...n-1])

4) If m2 is greater than m1, then median is present in one of the below two subarrays.

a) From m1 to last element of ar1 (ar1[|\_n/2\_|...n-1])

b) From first element of ar2 to m2 (ar2[0...|\_n/2\_|])

5) Repeat the above process until size of both the subarrays becomes 2. 6) If size of the two arrays is 2 then use below formula to get the median. Median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2

30 | 531 Program Creek

21 of 246 24 of 288

**7 Kth Largest Element in an Array**

Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

For example, given [3,2,1,5,6,4] and k = 2, return 5.

Note: You may assume k is always valid, 1 ≤ k ≤ array’s length. **7.1 Java Solution 1 - Sorting**

public int findKthLargest(int[] nums, int k) {

Arrays.sort(nums);

return nums[nums.length-k];

}

Time is O(nlog(n))

**7.2 Java Solution 2 - Quick Sort**

This problem can also be solve by using the quickselect approach, which is similar to quicksort.

public int findKthLargest(int[] nums, int k) {

if (k < 1 || nums == null) {

return 0;

}

return getKth(nums.length - k +1, nums, 0, nums.length - 1); }

public int getKth(int k, int[] nums, int start, int end) {

int pivot = nums[end];

int left = start;

int right = end;

while (true) {

while (nums[left] < pivot && left < right) {

left++;

}

31 | 531

22 of 246 25 of 288

7 Kth Largest Element in an Array

while (nums[right] >= pivot && right > left) {

right--;

}

if (left == right) {

break;

}

swap(nums, left, right);

}

swap(nums, left, end);

if (k == left + 1) {

return pivot;

} else if (k < left + 1) {

return getKth(k, nums, start, left - 1);

} else {

return getKth(k, nums, left + 1, end);

}

}

public void swap(int[] nums, int n1, int n2) {

int tmp = nums[n1];

nums[n1] = nums[n2];

nums[n2] = tmp;

}

Average case time is O(n), worst case time is O(n2ˆ).

**7.3 Java Solution 3 - Heap**

We can use a min heap to solve this problem. The heap stores the top k elements. Whenever the size is greater than k, delete the min. Time complexity is O(nlog(k)). Space complexity is O(k) for storing the top k numbers.

public int findKthLargest(int[] nums, int k) {

PriorityQueue<Integer> q = new PriorityQueue<Integer>(k);

for(int i: nums){

q.offer(i);

if(q.size()>k){

q.poll();

}

}

return q.peek();

}

32 | 531 Program Creek

23 of 246 26 of 288

**8 Wildcard Matching**

Implement wildcard pattern matching with support for ’?’ and ’\*’.

**8.1 Java Solution**

To understand this solution, you can use s="aab" and p="\*ab".

public boolean isMatch(String s, String p) {

int i = 0;

int j = 0;

int starIndex = -1;

int iIndex = -1;

while (i < s.length()) {

if (j < p.length() && (p.charAt(j) == ’?’ || p.charAt(j) == s.charAt(i))) {

++i;

++j;

} else if (j < p.length() && p.charAt(j) == ’\*’) {

starIndex = j;

iIndex = i;

j++;

} else if (starIndex != -1) {

j = starIndex + 1;

i = iIndex+1;

iIndex++;

} else {

return false;

}

}

while (j < p.length() && p.charAt(j) == ’\*’) {

++j;

}

return j == p.length();

}

33 | 531

24 of 246 27 of 288

**9 Regular Expression Matching in Java**

Implement regular expression matching with support for ’.’ and ’\*’.

’.’ Matches any single character. ’\*’ Matches zero or more of the preceding element. The matching should cover the entire input string (not partial).

The function prototype should be: bool isMatch(const char \*s, const char \*p) Some examples: isMatch("aa","a") return false isMatch("aa","aa") return true isMatch("aaa","aa") return false isMatch("aa", "a\*") return true isMatch("aa", ".\*") return true isMatch("ab", ".\*") return true isMatch("aab", "c\*a\*b") return true

**9.1 Analysis**

First of all, this is one of the most difficulty problems. It is hard to think through all different cases. The problem should be simplified to handle 2 basic cases:

• the second char of pattern is "\*"

• the second char of pattern is not "\*"

For the 1st case, if the first char of pattern is not ".", the first char of pattern and string should be the same. Then continue to match the remaining part.

For the 2nd case, if the first char of pattern is "." or first char of pattern == the first i char of string, continue to match the remaining part.

**9.2 Java Solution 1 (Short)**

The following Java solution is accepted.

public class Solution {

public boolean isMatch(String s, String p) {

if(p.length() == 0)

return s.length() == 0;

//p’s length 1 is special case

if(p.length() == 1 || p.charAt(1) != ’\*’){

if(s.length() < 1 || (p.charAt(0) != ’.’ && s.charAt(0) !=

p.charAt(0)))

return false;

return isMatch(s.substring(1), p.substring(1));

}else{

35 | 531

25 of 246 29 of 288

9 Regular Expression Matching in Java

int len = s.length();

int i = -1;

while(i<len && (i < 0 || p.charAt(0) == ’.’ || p.charAt(0) == s.charAt(i))){

if(isMatch(s.substring(i+1), p.substring(2)))

return true;

i++;

}

return false;

}

}

}

**9.3 Java Solution 2 (More Readable)**

public boolean isMatch(String s, String p) {

// base case

if (p.length() == 0) {

return s.length() == 0;

}

// special case

if (p.length() == 1) {

// if the length of s is 0, return false

if (s.length() < 1) {

return false;

}

//if the first does not match, return false

else if ((p.charAt(0) != s.charAt(0)) && (p.charAt(0) != ’.’)) { return false;

}

// otherwise, compare the rest of the string of s and p. else {

return isMatch(s.substring(1), p.substring(1));

}

}

// case 1: when the second char of p is not ’\*’

if (p.charAt(1) != ’\*’) {

if (s.length() < 1) {

return false;

}

if ((p.charAt(0) != s.charAt(0)) && (p.charAt(0) != ’.’)) {

36 | 531 Program Creek

26 of 246 30 of 288

9 Regular Expression Matching in Java

return false;

} else {

return isMatch(s.substring(1), p.substring(1));

}

}

// case 2: when the second char of p is ’\*’, complex case.

else {

//case 2.1: a char & ’\*’ can stand for 0 element

if (isMatch(s, p.substring(2))) {

return true;

}

//case 2.2: a char & ’\*’ can stand for 1 or more preceding element, //so try every sub string

int i = 0;

while (i<s.length() && (s.charAt(i)==p.charAt(0) || p.charAt(0)==’.’)){ if (isMatch(s.substring(i + 1), p.substring(2))) {

return true;

}

i++;

}

return false;

}

}

Program Creek 37 | 531

27 of 246 31 of 288

**10 Merge Intervals**

Problem:

Given a collection of intervals, merge all overlapping intervals.

For example,

Given [1,3],[2,6],[8,10],[15,18],

return [1,6],[8,10],[15,18].

**10.1 Thoughts of This Problem**

The key to solve this problem is defining a Comparator first to sort the arraylist of Intevals. And then merge some intervals.

The take-away message from this problem is utilizing the advantage of sorted list/ar ray.

**10.2 Java Solution**

class Interval {

int start;

int end;

Interval() {

start = 0;

end = 0;

}

Interval(int s, int e) {

start = s;

end = e;

}

}

public class Solution {

public ArrayList<Interval> merge(ArrayList<Interval> intervals) {

if (intervals == null || intervals.size() <= 1)

return intervals;

// sort intervals by using self-defined Comparator

39 | 531

28 of 246 33 of 288

10 Merge Intervals

Collections.sort(intervals, new IntervalComparator()); ArrayList<Interval> result = new ArrayList<Interval>();

Interval prev = intervals.get(0);

for (int i = 1; i < intervals.size(); i++) {

Interval curr = intervals.get(i);

if (prev.end >= curr.start) {

// merged case

Interval merged = new Interval(prev.start, Math.max(prev.end, curr.end));

prev = merged;

} else {

result.add(prev);

prev = curr;

}

}

result.add(prev);

return result;

}

}

class IntervalComparator implements Comparator<Interval> { public int compare(Interval i1, Interval i2) {

return i1.start - i2.start;

}

}

40 | 531 Program Creek

29 of 246 34 of 288

**11 Insert Interval**

Problem:

Given a set of non-overlapping & sorted intervals, insert a new interval into the intervals (merge if necessary).

Example 1:

Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].

Example 2:

Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].

This is because the new interval [4,9] overlaps with [3,5],[6,7],[8,10].

**11.1 Thoughts of This Problem**

Quickly summarize 3 cases. Whenever there is intersection, created a new interval.

**11.2 Java Solution**

41 | 531

30 of 246 35 of 288

11 Insert Interval

/\*\*

\* Definition for an interval.

\* public class Interval {

\* int start;

\* int end;

\* Interval() { start = 0; end = 0; }

\* Interval(int s, int e) { start = s; end = e; }

\* }

\*/

public class Solution {

public ArrayList<Interval> insert(ArrayList<Interval> intervals, Interval newInterval) {

ArrayList<Interval> result = new ArrayList<Interval>();

for(Interval interval: intervals){

if(interval.end < newInterval.start){

result.add(interval);

}else if(interval.start > newInterval.end){

result.add(newInterval);

newInterval = interval;

}else if(interval.end >= newInterval.start || interval.start <= newInterval.end){

newInterval = new Interval(Math.min(interval.start,

newInterval.start), Math.max(newInterval.end, interval.end)); }

}

result.add(newInterval);

return result;

}

}

42 | 531 Program Creek

31 of 246 36 of 288

**12 Two Sum**

Given an array of integers, find two numbers such that they add up to a specific target number.

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

For example:

Input: numbers={2, 7, 11, 15}, target=9

Output: index1=1, index2=2

**12.1 Naive Approach**

This problem is pretty straightforward. We can simply examine every possible pair of numbers in this integer array.

Time complexity in worst case: O(n2ˆ).

public static int[] twoSum(int[] numbers, int target) {

int[] ret = new int[2];

for (int i = 0; i < numbers.length; i++) {

for (int j = i + 1; j < numbers.length; j++) {

if (numbers[i] + numbers[j] == target) {

ret[0] = i + 1;

ret[1] = j + 1;

}

}

}

return ret;

}

Can we do better?

**12.2 Better Solution**

Use HashMap to store the target value.

public class Solution {

public int[] twoSum(int[] numbers, int target) {

HashMap<Integer, Integer> map = new HashMap<Integer, Integer>(); int[] result = new int[2];

43 | 531

32 of 246 37 of 288

12 Two Sum

for (int i = 0; i < numbers.length; i++) {

if (map.containsKey(numbers[i])) {

int index = map.get(numbers[i]);

result[0] = index+1 ;

result[1] = i+1;

break;

} else {

map.put(target - numbers[i], i);

}

}

return result;

}

}

Time complexity depends on the put and get operations of HashMap which is nor mally O(1).

Time complexity of this solution is O(n).

44 | 531 Program Creek

33 of 246 38 of 288

**13 Two Sum II Input array is sorted**

This problem is similar to Two Sum.

To solve this problem, we can use two points to scan the array from both sides. See Java solution below:

public int[] twoSum(int[] numbers, int target) {

if (numbers == null || numbers.length == 0)

return null;

int i = 0;

int j = numbers.length - 1;

while (i < j) {

int x = numbers[i] + numbers[j];

if (x < target) {

++i;

} else if (x > target) {

j--;

} else {

return new int[] { i + 1, j + 1 };

}

}

return null;

}

45 | 531

34 of 246 39 of 288

**14 Two Sum III Data structure design**

Design and implement a TwoSum class. It should support the following operations: add and find.

add - Add the number to an internal data structure. find - Find if there exists any pair of numbers which sum is equal to the value.

For example,

add(1);

add(3);

add(5);

find(4) -> true

find(7) -> false

**14.1 Java Solution**

Since the desired class need add and get operations, HashMap is a good option for this purpose.

public class TwoSum {

private HashMap<Integer, Integer> elements = new HashMap<Integer, Integer>();

public void add(int number) {

if (elements.containsKey(number)) {

elements.put(number, elements.get(number) + 1);

} else {

elements.put(number, 1);

}

}

public boolean find(int value) {

for (Integer i : elements.keySet()) {

int target = value - i;

if (elements.containsKey(target)) {

if (i == target && elements.get(target) < 2) {

continue;

}

return true;

}

}

return false;

47 | 531

35 of 246 41 of 288

14 Two Sum III Data structure design

}

}

48 | 531 Program Creek 36 of 246 42 of 288

**15 3Sum**

Problem:

Given an array S of n integers, are there elements a, b, c in S such that a + b + c = *0*? Find all unique triplets in the array which gives the sum of zero.

Note: Elements in a triplet (a,b,c) must be in non-descending order. (ie, a ≤ b ≤ c) The solution set must not contain duplicate triplets.

For example, given array S = {-1 0 1 2 -1 -4},

A solution set is:

(-1, 0, 1)

(-1, -1, 2)

**15.1 Naive Solution**

Naive solution is 3 loops, and this gives time complexity O(n3ˆ). Apparently this is not an acceptable solution, but a discussion can start from here.

public class Solution {

public ArrayList<ArrayList<Integer>> threeSum(int[] num) {

//sort array

Arrays.sort(num);

ArrayList<ArrayList<Integer>> result = new

ArrayList<ArrayList<Integer>>();

ArrayList<Integer> each = new ArrayList<Integer>();

for(int i=0; i<num.length; i++){

if(num[i] > 0) break;

for(int j=i+1; j<num.length; j++){

if(num[i] + num[j] > 0 && num[j] > 0) break;

for(int k=j+1; k<num.length; k++){

if(num[i] + num[j] + num[k] == 0) {

each.add(num[i]);

each.add(num[j]);

each.add(num[k]);

result.add(each);

each.clear();

}

49 | 531

37 of 246 43 of 288

15 3Sum

}

}

}

return result;

}

}

\* The solution also does not handle duplicates. Therefore, it is not only time ineffi cient, but also incorrect.

Result:

Submission Result: Output Limit Exceeded

**15.2 Better Solution**

A better solution is using two pointers instead of one. This makes time complexity of O(n2ˆ).

To avoid duplicate, we can take advantage of sorted arrays, i.e., move pointers by >1 to use same element only once.

public ArrayList<ArrayList<Integer>> threeSum(int[] num) {

ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

if (num.length < 3)

return result;

// sort array

Arrays.sort(num);

for (int i = 0; i < num.length - 2; i++) {

// //avoid duplicate solutions

if (i == 0 || num[i] > num[i - 1]) {

int negate = -num[i];

int start = i + 1;

int end = num.length - 1;

while (start < end) {

//case 1

if (num[start] + num[end] == negate) {

ArrayList<Integer> temp = new ArrayList<Integer>();

temp.add(num[i]);

temp.add(num[start]);

temp.add(num[end]);

result.add(temp);

50 | 531 Program Creek

38 of 246 44 of 288

15 3Sum

start++;

end--;

//avoid duplicate solutions

while (start < end && num[end] == num[end + 1])

end--;

while (start < end && num[start] == num[start - 1])

start++;

//case 2

} else if (num[start] + num[end] < negate) {

start++;

//case 3

} else {

end--;

}

}

}

}

return result;

}

Program Creek 51 | 531

39 of 246 45 of 288

**16 4Sum**

Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target. Note: Elements in a quadruplet (a,b,c,d) must be in non-descending order. (ie, a ≤ b ≤ c ≤ d) The solution set must not contain duplicate quadruplets.

For example, given array S = {1 0 -1 0 -2 2}, and target = 0.

A solution set is:

(-1, 0, 0, 1)

(-2, -1, 1, 2)

(-2, 0, 0, 2)

**16.1 Thoughts**

A typical k-sum problem. Time is N to the power of (k-1).

**16.2 Java Solution**

public ArrayList<ArrayList<Integer>> fourSum(int[] num, int target) { Arrays.sort(num);

HashSet<ArrayList<Integer>> hashSet = new HashSet<ArrayList<Integer>>(); ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

for (int i = 0; i < num.length; i++) {

for (int j = i + 1; j < num.length; j++) {

int k = j + 1;

int l = num.length - 1;

while (k < l) {

int sum = num[i] + num[j] + num[k] + num[l];

if (sum > target) {

l--;

} else if (sum < target) {

k++;

} else if (sum == target) {

ArrayList<Integer> temp = new ArrayList<Integer>();

temp.add(num[i]);

53 | 531

40 of 246 47 of 288

16 4Sum

temp.add(num[j]);

temp.add(num[k]);

temp.add(num[l]);

if (!hashSet.contains(temp)) {

hashSet.add(temp);

result.add(temp);

}

k++;

l--;

}

}

}

}

return result;

}

Here is the hashCode method of ArrayList. It makes sure that if all elements of two lists are the same, then the hash code of the two lists will be the same. Since each element in the ArrayList is Integer, same integer has same hash code.

int hashCode = 1;

Iterator<E> i = list.iterator();

while (i.hasNext()) {

E obj = i.next();

hashCode = 31\*hashCode + (obj==null ? 0 : obj.hashCode());

}

54 | 531 Program Creek

41 of 246 48 of 288

**17 3Sum Closest**

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

For example, given array S = {-1 2 1 -4}, and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

**17.1 Analysis**

This problem is similar to 2 Sum. This kind of problem can be solved by using a similar approach, i.e., two pointers from both left and right.

**17.2 Java Solution**

public int threeSumClosest(int[] nums, int target) {

int min = Integer.MAX\_VALUE;

int result = 0;

Arrays.sort(nums);

for (int i = 0; i < nums.length; i++) {

int j = i + 1;

int k = nums.length - 1;

while (j < k) {

int sum = nums[i] + nums[j] + nums[k];

int diff = Math.abs(sum - target);

if(diff == 0) return sum;

if (diff < min) {

min = diff;

result = sum;

}

if (sum <= target) {

j++;

} else {

k--;

}

}

55 | 531

42 of 246 49 of 288

17 3Sum Closest

}

return result;

}

Time Complexity is O(n2ˆ).

56 | 531 Program Creek 43 of 246 50 of 288

**18 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.

**18.1 Analysis**

The following cases should be considered for this problem:

1. null or empty string

2. white spaces

3. +/- sign

4. calculate real value

5. handle min & max

**18.2 Java Solution**

public int atoi(String str) {

if (str == null || str.length() < 1)

return 0;

// trim white spaces

str = str.trim();

char flag = ’+’;

// check negative or positive

int i = 0;

if (str.charAt(0) == ’-’) {

flag = ’-’;

i++;

} else if (str.charAt(0) == ’+’) {

i++;

}

// use double to store result

double result = 0;

// calculate value

while (str.length() > i && str.charAt(i) >= ’0’ && str.charAt(i) <= ’9’) { result = result \* 10 + (str.charAt(i) - ’0’);

57 | 531

44 of 246 51 of 288

18 String to Integer (atoi)

i++;

}

if (flag == ’-’)

result = -result;

// handle max and min

if (result > Integer.MAX\_VALUE)

return Integer.MAX\_VALUE;

if (result < Integer.MIN\_VALUE)

return Integer.MIN\_VALUE;

return (int) result;

}

58 | 531 Program Creek 45 of 246 52 of 288

**19 Merge Sorted Array**

Given two sorted integer arrays A and B, merge B into A as one sorted array.

Note: You may assume that A has enough space to hold additional elements from B. The number of elements initialized in A and B are m and n respectively.

**19.1 Analysis**

The key to solve this problem is moving element of A and B backwards. If B has some elements left after A is done, also need to handle that case.

The takeaway message from this problem is that the loop condition. This kind of condition is also used for merging two sorted linked list.

**19.2 Java Solution 1**

public class Solution {

public void merge(int A[], int m, int B[], int n) {

while(m > 0 && n > 0){

if(A[m-1] > B[n-1]){

A[m+n-1] = A[m-1];

m--;

}else{

A[m+n-1] = B[n-1];

n--;

}

}

while(n > 0){

A[m+n-1] = B[n-1];

n--;

}

}

}

**19.3 Java Solution 2**

The loop condition also can use m+n like the following.

59 | 531

46 of 246 53 of 288

19 Merge Sorted Array

public void merge(int A[], int m, int B[], int n) { int i = m - 1;

int j = n - 1;

int k = m + n - 1;

while (k >= 0) {

if (j < 0 || (i >= 0 && A[i] > B[j]))

A[k--] = A[i--];

else

A[k--] = B[j--];

}

}

60 | 531 Program Creek 47 of 246 54 of 288

**20 Valid Parentheses**

Given a string containing just the characters ’(’, ’)’, ’’, ’’, ’[’ and ’]’, determine if the input string is valid. The brackets must close in the correct order, "()" and "()[]" are all valid but "(]" and "([)]" are not.

**20.1 Analysis**

A typical problem which can be solved by using a stack data structure. **20.2 Java Solution**

public static boolean isValid(String s) {

HashMap<Character, Character> map = new HashMap<Character, Character>(); map.put(’(’, ’)’);

map.put(’[’, ’]’);

map.put(’{’, ’}’);

Stack<Character> stack = new Stack<Character>();

for (int i = 0; i < s.length(); i++) {

char curr = s.charAt(i);

if (map.keySet().contains(curr)) {

stack.push(curr);

} else if (map.values().contains(curr)) {

if (!stack.empty() && map.get(stack.peek()) == curr) {

stack.pop();

} else {

return false;

}

}

}

return stack.empty();

}

61 | 531

48 of 246 55 of 288

**21 Longest Valid Parentheses**

Given a string containing just the characters ’(’ and ’)’, find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2. An other example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

**21.1 Analysis**

This problem is similar with Valid Parentheses, which can be solved by using a stack. **21.2 Java Solution**

public static int longestValidParentheses(String s) {

Stack<int[]> stack = new Stack<int[]>();

int result = 0;

for(int i=0; i<=s.length()-1; i++){

char c = s.charAt(i);

if(c==’(’){

int[] a = {i,0};

stack.push(a);

}else{

if(stack.empty()||stack.peek()[1]==1){

int[] a = {i,1};

stack.push(a);

}else{

stack.pop();

int currentLen=0;

if(stack.empty()){

currentLen = i+1;

}else{

currentLen = i-stack.peek()[0];

}

result = Math.max(result, currentLen);

}

}

}

return result;

63 | 531

49 of 246 57 of 288

21 Longest Valid Parentheses

}

64 | 531 Program Creek 50 of 246 58 of 288

**22 Implement strStr()**

Problem:

Implement strStr(). Returns the index of the first occurrence of needle in haystack, or -*1* if needle is not part of haystack.

**22.1 Java Solution 1 - Naive**

public int strStr(String haystack, String needle) {

if(haystack==null || needle==null)

return 0;

if(needle.length() == 0)

return 0;

for(int i=0; i<haystack.length(); i++){

if(i + needle.length() > haystack.length())

return -1;

int m = i;

for(int j=0; j<needle.length(); j++){

if(needle.charAt(j)==haystack.charAt(m)){

if(j==needle.length()-1)

return i;

m++;

}else{

break;

}

}

}

return -1;

}

**22.2 Java Solution 2 - KMP**

Check out this article to understand KMP algorithm.

public int strStr(String haystack, String needle) {

if(haystack==null || needle==null)

65 | 531

51 of 246 59 of 288

22 Implement strStr()

return 0;

int h = haystack.length();

int n = needle.length();

if (n > h)

return -1;

if (n == 0)

return 0;

int[] next = getNext(needle);

int i = 0;

while (i <= h - n) {

int success = 1;

for (int j = 0; j < n; j++) {

if (needle.charAt(0) != haystack.charAt(i)) {

success = 0;

i++;

break;

} else if (needle.charAt(j) != haystack.charAt(i + j)) { success = 0;

i = i + j - next[j - 1];

break;

}

}

if (success == 1)

return i;

}

return -1;

}

//calculate KMP array

public int[] getNext(String needle) {

int[] next = new int[needle.length()];

next[0] = 0;

for (int i = 1; i < needle.length(); i++) {

int index = next[i - 1];

while (index > 0 && needle.charAt(index) != needle.charAt(i)) { index = next[index - 1];

}

if (needle.charAt(index) == needle.charAt(i)) {

next[i] = next[i - 1] + 1;

} else {

next[i] = 0;

}

}

66 | 531 Program Creek

52 of 246 60 of 288

22 Implement strStr()

return next;

}

Program Creek 67 | 531

53 of 246 61 of 288

**23 Minimum Size Subarray Sum**

Given an array of n positive integers and a positive integer s, find the minimal length of a subarray of which the sum ≥ s. If there isn’t one, return 0 instead. For example, given the array [2,3,1,2,4,3] and s = 7, the subarray [4,3] has the minimal length of 2 under the problem constraint.

**23.1 Analysis**

We can use 2 points to mark the left and right boundaries of the sliding window. When the sum is greater than the target, shift the left pointer; when the sum is less than the target, shift the right pointer.

**23.2 Java Solution 1**

A simple sliding window solution.

public int minSubArrayLen(int s, int[] nums) {

if(nums==null || nums.length==1)

return 0;

int result = nums.length;

int start=0;

int sum=0;

int i=0;

boolean exists = false;

while(i<=nums.length){

if(sum>=s){

exists=true; //mark if there exists such a subarray

if(start==i-1){

return 1;

}

result = Math.min(result, i-start);

sum=sum-nums[start];

start++;

}else{

if(i==nums.length)

break;

69 | 531

54 of 246 63 of 288

23 Minimum Size Subarray Sum

sum = sum+nums[i];

i++;

}

}

if(exists)

return result;

else

return 0;

}

**23.3 Deprecated Java Solution**

This solution works but it is less readable.

public int minSubArrayLen(int s, int[] nums) { if(nums == null || nums.length == 0){

return 0;

}

if(nums.length == 1 && nums[0] < s){

return 0;

}

// initialize min length to be the input array length int result = nums.length;

int i = 0;

int sum = nums[0];

for(int j=0; j<nums.length; ){

if(i==j){

if(nums[i]>=s){

return 1; //if single elem is large enough

}else{

j++;

if(j<nums.length){

sum = sum + nums[j];

}else{

return result;

}

}

}else{

//if sum is large enough, move left cursor

if(sum >= s){

result = Math.min(j-i+1, result);

sum = sum - nums[i];

i++;

70 | 531 Program Creek 55 of 246 64 of 288

23 Minimum Size Subarray Sum

//if sum is not large enough, move right cursor

}else{

j++;

if(j<nums.length){

sum = sum + nums[j];

}else{

if(i==0){

return 0;

}else{

return result;

}

}

}

}

}

return result;

}

Program Creek 71 | 531

56 of 246 65 of 288

**24 Search Insert Position**

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order. You may assume no duplicates in the array.

Here are few examples.

[1,3,5,6], 5 -> 2

[1,3,5,6], 2 -> 1

[1,3,5,6], 7 -> 4

[1,3,5,6], 0 -> 0

**24.1 Solution 1**

Naively, we can just iterate the array and compare target with ith and (i+1)th element. Time complexity is O(n)

public class Solution {

public int searchInsert(int[] A, int target) {

if(A==null) return 0;

if(target <= A[0]) return 0;

for(int i=0; i<A.length-1; i++){

if(target > A[i] && target <= A[i+1]){

return i+1;

}

}

return A.length;

}

}

**24.2 Solution 2**

This also looks like a binary search problem. We should try to make the complexity to be O(log(n)).

public class Solution {

public int searchInsert(int[] A, int target) {

73 | 531

57 of 246 67 of 288

24 Search Insert Position

if(A==null||A.length==0)

return 0;

return searchInsert(A,target,0,A.length-1);

}

public int searchInsert(int[] A, int target, int start, int end){ int mid=(start+end)/2;

if(target==A[mid])

return mid;

else if(target<A[mid])

return start<mid?searchInsert(A,target,start,mid-1):start; else

return end>mid?searchInsert(A,target,mid+1,end):(end+1); }

}

74 | 531 Program Creek

58 of 246 68 of 288

**25 Longest Consecutive Sequence**

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example, given [100, 4, 200, 1, 3, 2], the longest consecutive elements sequence should be [1, 2, 3, 4]. Its length is 4.

Your algorithm should run in O(n) complexity.

**25.1 Analysis**

Because it requires O(n) complexity, we can not solve the problem by sorting the array first. Sorting takes at least O(nlogn) time.

**25.2 Java Solution**

We can use a HashSet to add and remove elements. HashSet is implemented by using a hash table. Elements are not ordered. The add, remove and contains methods have constant time complexity O(1).

public static int longestConsecutive(int[] num) {

// if array is empty, return 0

if (num.length == 0) {

return 0;

}

Set<Integer> set = new HashSet<Integer>();

int max = 1;

for (int e : num)

set.add(e);

for (int e : num) {

int left = e - 1;

int right = e + 1;

int count = 1;

while (set.contains(left)) {

count++;

set.remove(left);

left--;

}

75 | 531

59 of 246 69 of 288

25 Longest Consecutive Sequence

while (set.contains(right)) {

count++;

set.remove(right);

right++;

}

max = Math.max(count, max);

}

return max;

}

After an element is checked, it should be removed from the set. Otherwise, time complexity would be O(mn) in which m is the average length of all consecutive se quences.

To clearly see the time complexity, I suggest you use some simple examples and manually execute the program. For example, given an array 1,2,4,5,3, the program time is m. m is the length of longest consecutive sequence.

76 | 531 Program Creek

60 of 246 70 of 288

**26 Valid Palindrome**

Given a string, determine if it is a palindrome, considering only alphanumeric charac ters and ignoring cases.

For example, "Red rum, sir, is murder" is a palindrome, while "Programcreek is awesome" is not.

Note: Have you consider that the string might be empty? This is a good question to ask during an interview.

For the purpose of this problem, we define empty string as valid palindrome.

**26.1 Thoughts**

From start and end loop though the string, i.e., char array. If it is not alpha or num ber, increase or decrease pointers. Compare the alpha and numeric characters. The solution below is pretty straightforward.

**26.2 Java Solution 1 - Naive**

public class Solution {

public boolean isPalindrome(String s) {

if(s == null) return false;

if(s.length() < 2) return true;

char[] charArray = s.toCharArray();

int len = s.length();

int i=0;

int j=len-1;

while(i<j){

char left, right;

while(i<len-1 && !isAlpha(left) && !isNum(left)){

i++;

left = charArray[i];

}

while(j>0 && !isAlpha(right) && !isNum(right)){

j--;

77 | 531

61 of 246 71 of 288

26 Valid Palindrome

right = charArray[j];

}

if(i >= j)

break;

left = charArray[i];

right = charArray[j];

if(!isSame(left, right)){

return false;

}

i++;

j--;

}

return true;

}

public boolean isAlpha(char a){

if((a >= ’a’ && a <= ’z’) || (a >= ’A’ && a <= ’Z’)){ return true;

}else{

return false;

}

}

public boolean isNum(char a){

if(a >= ’0’ && a <= ’9’){

return true;

}else{

return false;

}

}

public boolean isSame(char a, char b){

if(isNum(a) && isNum(b)){

return a == b;

}else if(Character.toLowerCase(a) == Character.toLowerCase(b)){ return true;

}else{

return false;

}

}

}

78 | 531 Program Creek

62 of 246 72 of 288

26 Valid Palindrome

**26.3 Java Solution 2 - Using Stack**

This solution removes the special characters first. (Thanks to Tia)

public boolean isPalindrome(String s) {

s = s.replaceAll("[^a-zA-Z0-9]","").toLowerCase();

int len = s.length();

if (len < 2)

return true;

Stack<Character> stack = new Stack<Character>();

int index = 0;

while (index < len / 2) {

stack.push(s.charAt(index));

index++;

}

if (len % 2 == 1)

index++;

while (index < len) {

if (stack.empty())

return false;

char temp = stack.pop();

if (s.charAt(index) != temp)

return false;

else

index++;

}

return true;

}

**26.4 Java Solution 3 - Using Two Pointers**

In the discussion below, April and Frank use two pointers to solve this problem. This solution looks really simple.

public class ValidPalindrome {

public static boolean isValidPalindrome(String s){

if(s==null||s.length()==0) return false;

s = s.replaceAll("[^a-zA-Z0-9]","").toLowerCase();

System.out.println(s);

Program Creek 79 | 531

63 of 246 73 of 288

26 Valid Palindrome

for(int i = 0; i < s.length() ; i++){

if(s.charAt(i) != s.charAt(s.length() - 1 - i)){ return false;

}

}

return true;

}

public static void main(String[] args) {

String str = "A man, a plan, a canal: Panama";

System.out.println(isValidPalindrome(str)); }

}

80 | 531 Program Creek 64 of 246 74 of 288

**27 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)

P A H N

A P L S I I G

Y I R

And then read line by line: "PAHNAPLSIIGYIR" Write the a method convert("PAYPALISHIRING", 3) which returns "PAHNAPLSIIGYIR".

**27.1 Java Solution**

public String convert(String s, int numRows) {

if (numRows == 1)

return s;

StringBuilder sb = new StringBuilder();

// step

int step = 2 \* numRows - 2;

for (int i = 0; i < numRows; i++) {

//first & last rows

if (i == 0 || i == numRows - 1) {

for (int j = i; j < s.length(); j = j + step) {

sb.append(s.charAt(j));

}

//middle rows

} else {

int j = i;

boolean flag = true;

int step1 = 2 \* (numRows - 1 - i);

int step2 = step - step1;

while (j < s.length()) {

sb.append(s.charAt(j));

if (flag)

j = j + step1;

else

j = j + step2;

flag = !flag;

}

81 | 531

65 of 246 75 of 288

27 ZigZag Conversion

}

}

return sb.toString();

}

82 | 531 Program Creek 66 of 246 76 of 288

**28 Add Binary**

Given two binary strings, return their sum (also a binary string). For example, a = "11", b = "1", the return is "100".

**28.1 Java Solution**

Very simple, nothing special. Note how to convert a character to an int.

public String addBinary(String a, String b) {

if(a==null || a.length()==0)

return b;

if(b==null || b.length()==0)

return a;

int pa = a.length()-1;

int pb = b.length()-1;

int flag = 0;

StringBuilder sb = new StringBuilder();

while(pa >= 0 || pb >=0){

int va = 0;

int vb = 0;

if(pa >= 0){

va = a.charAt(pa)==’0’? 0 : 1;

pa--;

}

if(pb >= 0){

vb = b.charAt(pb)==’0’? 0: 1;

pb--;

}

int sum = va + vb + flag;

if(sum >= 2){

sb.append(String.valueOf(sum-2));

flag = 1;

}else{

flag = 0;

sb.append(String.valueOf(sum));

}

}

if(flag == 1){

83 | 531

67 of 246 77 of 288

28 Add Binary

sb.append("1");

}

String reversed = sb.reverse().toString(); return reversed;

}

84 | 531 Program Creek 68 of 246 78 of 288

**29 Length of Last Word**

Given a string s consists of upper/lower-case alphabets and empty space characters ’ ’, return the length of last word in the string. If the last word does not exist, return 0.

**29.1 Java Solution**

Very simple question. We just need a flag to mark the start of letters from the end. If a letter starts and the next character is not a letter, return the length.

public int lengthOfLastWord(String s) {

if(s==null || s.length() == 0)

return 0;

int result = 0;

int len = s.length();

boolean flag = false;

for(int i=len-1; i>=0; i--){

char c = s.charAt(i);

if((c>=’a’ && c<=’z’) || (c>=’A’ && c<=’Z’)){

flag = true;

result++;

}else{

if(flag)

return result;

}

}

return result;

}

85 | 531

69 of 246 79 of 288

**30 Triangle**

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

[

[2],

[3,4],

[6,5,7],

[4,1,8,3]

]

The minimum path sum from top to bottom is 11 (i.e., 2 + 3 + 5 + 1 = 11). Note: Bonus point if you are able to do this using only O(n) extra space, where n is the total number of rows in the triangle.

**30.1 Top-Down Approach (Wrong Answer!)**

This solution gets wrong answer! I will try to make it work later.

public class Solution {

public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {

int[] temp = new int[triangle.size()];

int minTotal = Integer.MAX\_VALUE;

for(int i=0; i< temp.length; i++){

temp[i] = Integer.MAX\_VALUE;

}

if (triangle.size() == 1) {

return Math.min(minTotal, triangle.get(0).get(0));

}

int first = triangle.get(0).get(0);

for (int i = 0; i < triangle.size() - 1; i++) {

for (int j = 0; j <= i; j++) {

int a, b;

if(i==0 && j==0){

87 | 531

70 of 246 81 of 288

30 Triangle

a = first + triangle.get(i + 1).get(j);

b = first + triangle.get(i + 1).get(j + 1);

}else{

a = temp[j] + triangle.get(i + 1).get(j);

b = temp[j] + triangle.get(i + 1).get(j + 1);

}

temp[j] = Math.min(a, temp[j]);

temp[j + 1] = Math.min(b, temp[j + 1]);

}

}

for (int e : temp) {

if (e < minTotal)

minTotal = e;

}

return minTotal;

}

}

**30.2 Bottom-Up (Good Solution)**

We can actually start from the bottom of the triangle.

public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) { int[] total = new int[triangle.size()];

int l = triangle.size() - 1;

for (int i = 0; i < triangle.get(l).size(); i++) {

total[i] = triangle.get(l).get(i);

}

// iterate from last second row

for (int i = triangle.size() - 2; i >= 0; i--) {

for (int j = 0; j < triangle.get(i + 1).size() - 1; j++) { total[j] = triangle.get(i).get(j) + Math.min(total[j], total[j + 1]); }

}

return total[0];

}

88 | 531 Program Creek

71 of 246 82 of 288

**31 Contains Duplicate**

Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

**31.1 Java Solution**

public boolean containsDuplicate(int[] nums) {

if(nums==null || nums.length==0)

return false;

HashSet<Integer> set = new HashSet<Integer>();

for(int i: nums){

if(!set.add(i)){

return true;

}

}

return false;

}

89 | 531

72 of 246 83 of 288

**32 Contains Duplicate II**

Given an array of integers and an integer k, return true if and only if there are two distinct indices i and j in the array such that nums[i] = nums[j] and the difference between i and j is at most k.

**32.1 Java Solution 1**

public boolean containsNearbyDuplicate(int[] nums, int k) {

HashMap<Integer, Integer> map = new HashMap<Integer, Integer>(); int min = Integer.MAX\_VALUE;

for(int i=0; i<nums.length; i++){

if(map.containsKey(nums[i])){

int preIndex = map.get(nums[i]);

int gap = i-preIndex;

min = Math.min(min, gap);

}

map.put(nums[i], i);

}

if(min <= k){

return true;

}else{

return false;

}

}

**32.2 Java Solution 2 - Simplified**

public boolean containsNearbyDuplicate(int[] nums, int k) {

HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

for(int i=0; i<nums.length; i++){

if(map.containsKey(nums[i])){

int pre = map.get(nums[i]);

if(i-pre<=k)

return true;

}

91 | 531

73 of 246 85 of 288

32 Contains Duplicate II

map.put(nums[i], i);

}

return false;

}

92 | 531 Program Creek 74 of 246 86 of 288

**33 Contains Duplicate III**

Given an array of integers, find out whether there are two distinct indices i and j in the array such that the difference between nums[i] and nums[j] is at most t and the difference between i and j is at most k.

**33.1 Java Solution 1**

This solution uses a treeset.

The time complexity is O(nlog(k)).

public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) { if (k < 1 || t < 0)

return false;

TreeSet<Integer> set = new TreeSet<Integer>();

for (int i = 0; i < nums.length; i++) {

int c = nums[i];

if ((set.floor(c) != null && c <= set.floor(c) + t)

|| (set.ceiling(c) != null && c >= set.ceiling(c) -t))

return true;

set.add(c);

if (i >= k)

set.remove(nums[i - k]);

}

return false;

}

93 | 531

75 of 246 87 of 288

33 Contains Duplicate III

**33.2 Java Solution 2**

Another solution that is easier to understand.

public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) { if (k < 1 || t < 0)

return false;

SortedSet<Long> set = new TreeSet<Long>();

for (int j = 0; j < nums.length; j++) {

long leftBoundary = (long) nums[j] - t;

long rightBoundary = (long) nums[j] + t + 1;

SortedSet<Long> subSet = set.subSet(leftBoundary, rightBoundary);

if (!subSet.isEmpty())

return true;

set.add((long) nums[j]);

if (j >= k) {

set.remove((long) nums[j - k]);

}

}

return false;

}

94 | 531 Program Creek

76 of 246 88 of 288

**34 Remove Duplicates from Sorted Array**

Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length. Do not allocate extra space for another array, you must do this in place with constant memory.

For example, given input array A = [1,1,2], your function should return length = 2, and A is now [1,2].

**34.1 Thoughts**

The problem is pretty straightforward. It returns the length of array with unique elements, but the original array need to be changed also. This problem should be reviewed with Remove Duplicates from Sorted Array II.

**34.2 Solution 1**

// Manipulate original array

public static int removeDuplicatesNaive(int[] A) {

if (A.length < 2)

return A.length;

int j = 0;

int i = 1;

while (i < A.length) {

if (A[i] == A[j]) {

i++;

} else {

j++;

A[j] = A[i];

i++;

}

}

return j + 1;

}

This method returns the number of unique elements, but does not change the orig inal array correctly. For example, if the input array is 1, 2, 2, 3, 3, the array will be changed to 1, 2, 3, 3, 3. The correct result should be 1, 2, 3. Because array’s size can

95 | 531

77 of 246 89 of 288

34 Remove Duplicates from Sorted Array

not be changed once created, there is no way we can return the original array with correct results.

**34.3 Solution 2**

// Create an array with all unique elements

public static int[] removeDuplicates(int[] A) {

if (A.length < 2)

return A;

int j = 0;

int i = 1;

while (i < A.length) {

if (A[i] == A[j]) {

i++;

} else {

j++;

A[j] = A[i];

i++;

}

}

int[] B = Arrays.copyOf(A, j + 1);

return B;

}

public static void main(String[] args) {

int[] arr = { 1, 2, 2, 3, 3 };

arr = removeDuplicates(arr);

System.out.println(arr.length);

}

In this method, a new array is created and returned.

**34.4 Solution 3**

If we only want to count the number of unique elements, the following method is good enough.

// Count the number of unique elements

public static int countUnique(int[] A) {

int count = 0;

for (int i = 0; i < A.length - 1; i++) {

if (A[i] == A[i + 1]) {

count++;

96 | 531 Program Creek

78 of 246 90 of 288

34 Remove Duplicates from Sorted Array

}

}

return (A.length - count);

}

public static void main(String[] args) {

int[] arr = { 1, 2, 2, 3, 3 };

int size = countUnique(arr);

System.out.println(size);

}

Program Creek 97 | 531

79 of 246 91 of 288

**35 Remove Duplicates from Sorted Array II**

Follow up for "Remove Duplicates": What if duplicates are allowed at most twice? For example, given sorted array A = [1,1,1,2,2,3], your function should return length = 5, and A is now [1,1,2,2,3].

**35.1 Naive Approach**

Given the method signature "public int removeDuplicates(int[] A)", it seems that we should write a method that returns a integer and that’s it. After typing the following solution:

public class Solution {

public int removeDuplicates(int[] A) {

if(A == null || A.length == 0)

return 0;

int pre = A[0];

boolean flag = false;

int count = 0;

for(int i=1; i<A.length; i++){

int curr = A[i];

if(curr == pre){

if(!flag){

flag = true;

continue;

}else{

count++;

}

}else{

pre = curr;

flag = false;

}

}

return A.length - count;

}

}

99 | 531

80 of 246 93 of 288