Coding Interview in Java

Program Creek

Contents

Every title in the PDF is linked back to the original blog. When it is clicked, it opens the original post in your browser. If you want to discuss any problem, please go to the post and leave your comment there.

I'm not an expert and some solutions may not be optimal. So please leave your comment if you see any problem or have a better solution. I will reply your comment as soon as I can.

This collection is updated from time to time. Please check out this link for the latest version: http://www.programcreek.com/10-algorithms-for-coding-interview/

1 Remove Duplicates from Sorted Array

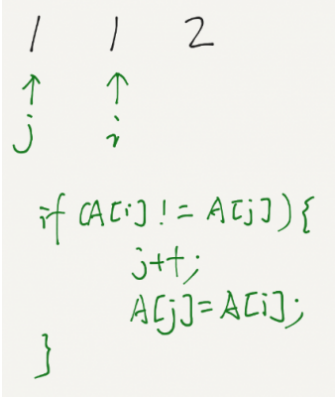
Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length. Do not allocate extra space for another array, you must do this in place with constant memory.

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

1.1 Analysis

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

1.2 Java Solution




```
public static int removeDuplicates(int[] A) {
  if (A.length < 2)
    return A.length;

  int j = 0;
  int i = 1;

  while (i < A.length) {
    if (A[i] != A[j]) {
      j++;
      A[j] = A[i];
    }

          i++;
  }

  return j + 1;
}
```

Note that we only care about the first unique part of the original array. So it is ok if input array is 1, 2, 2, 3, 3, the array is changed to 1, 2, 3, 3, 3.

2 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].

So this problem also requires in-place array manipulation.

2.1 Java Solution 1

We can not change the given array's size, so we only change the first k elements of the array which has duplicates removed.

```
public int removeDuplicates(int[] nums) {
   if(nums==null){
      return 0;
   }
   if(nums.length<3){
      return nums.length;
   }

   int i=0;
   int j=1;
/*

       i, j  1 1 1 2 2 3
step1  0 1    i j
step2  1 2    i  j
step3  1 3      i  j
step4  2 4        i  j

*/
   while(j<nums.length){
      if(nums[j]==nums[i]){
         if(i==0){
            i++;
            j++;
         }else if(nums[i]==nums[i-1]){
            j++;
         }else{
            i++;
            nums[i]=nums[j];
            j++;
         }
      }else{
         i++;
         nums[i]=nums[j];
         j++;
      }
   }

   return i+1;
}
```

The problem with this solution is that there are 4 cases to handle. If we shift our two points to right by 1 element, the solution can be simplified as the Solution 2.

2.2 Java Solution 2

```
public int removeDuplicates(int[] nums) {
   if(nums==null){
      return 0;
   }

   if (nums.length <= 2){
      return nums.length;
   }
/*
1,1,1,2,2,3
  i j
*/
   int i = 1; // point to previous
   int j = 2; // point to current

   while (j < nums.length) {
      if (nums[j] == nums[i] && nums[j] == nums[i - 1]) {
         j++;
      } else {
         i++;
         nums[i] = nums[j];
         j++;
      }
   }

   return i + 1;
}
```

3 Remove Element

Given an array and a value, remove all instances of that value in place and return the new length. (Note: The order of elements can be changed. It doesn't matter what you leave beyond the new length.)

3.1 Java Solution

This problem can be solve by using two indices.

```
public int removeElement(int[] A, int elem) {
   int i=0;
   int j=0;

   while(j < A.length){
      if(A[j] != elem){
         A[i] = A[j];
         i++;
      }

      j++;
   }

   return i;
}
```

4 Move Zeroes

Given an array nums, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements.

For example, given nums = [0, 1, 0, 3, 12], after calling your function, nums should be [1, 3, 12, 0, 0].

4.1 Java Solution 2

We can use the similar code that is used to solve Remove Duplicates from Sorted Array I, II, Remove Element.

```
public void moveZeroes(int[] nums) {
   int i=0;
   int j=0;

   while(j<nums.length){
      if(nums[j]==0){
         j++;
      }else{
         nums[i]=nums[j];
         i++;
         j++;
      }
   }

   while(i<nums.length){
      nums[i]=0;
      i++;
   }
}
```

5 Candy

There are N children standing in a line. Each child is assigned a rating value. You are giving candies to these children subjected to the following requirements:

1. Each child must have at least one candy. 2. Children with a higher rating get more candies than their neighbors.

What is the minimum candies you must give?

5.1 Analysis

This problem can be solved in O(n) time.

We can always assign a neighbor with 1 more if the neighbor has higher a rating value. However, to get the minimum total number, we should always start adding 1s in the ascending order. We can solve this problem by scanning the array from both sides. First, scan the array from left to right, and assign values for all the ascending pairs. Then scan from right to left and assign values to descending pairs.

This problem is similar to Trapping Rain Water.

5.2 Java Solution

```
public int candy(int[] ratings) {
  if (ratings == null || ratings.length == 0) {
    return 0;
  }

  int[] candies = new int[ratings.length];
  candies[0] = 1;

  //from let to right
  for (int i = 1; i < ratings.length; i++) {
    if (ratings[i] > ratings[i - 1]) {
      candies[i] = candies[i - 1] + 1;
    } else {
      // if not ascending, assign 1
      candies[i] = 1;
    }
  }

  int result = candies[ratings.length - 1];

  //from right to left
  for (int i = ratings.length - 2; i >= 0; i--) {
    int cur = 1;
    if (ratings[i] > ratings[i + 1]) {
      cur = candies[i + 1] + 1;
    }

    result += Math.max(cur, candies[i]);
    candies[i] = cur;
  }
```

```
  return result;
}
```

6 Trapping Rain Water

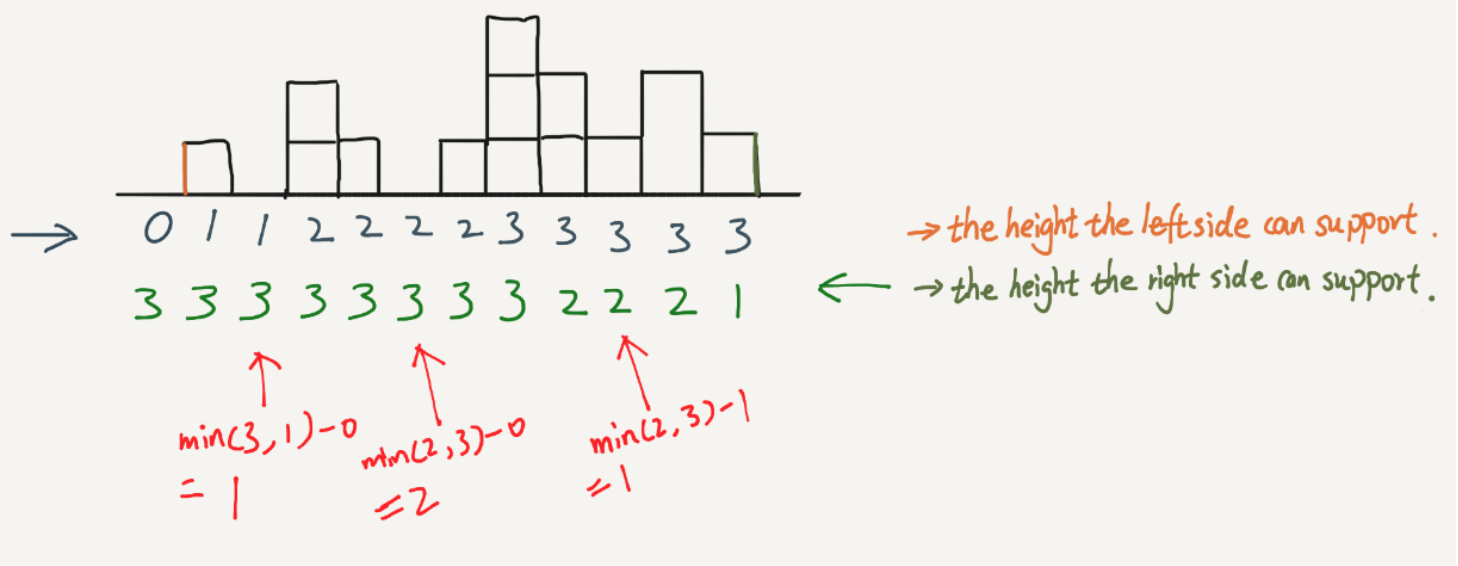
Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

For example, given [0,1,0,2,1,0,1,3,2,1,2,1], return 6.

6.1 Analysis

This problem is similar to Candy. It can be solve by scanning from both sides and then get the total.

6.2 Java Solution




```
public int trap(int[] height) {
   int result = 0;

   if(height==null || height.length<=2)
      return result;

   int left[] = new int[height.length];
   int right[]= new int[height.length];

   //scan from left to right
   int max = height[0];
   left[0] = height[0];
   for(int i=1; i<height.length; i++){
      if(height[i]<max){
         left[i]=max;
```

```
      }else{
         left[i]=height[i];
         max = height[i];
      }
   }

   //scan from right to left
   max = height[height.length-1];
   right[height.length-1]=height[height.length-1];
   for(int i=height.length-2; i>=0; i--){
      if(height[i]<max){
         right[i]=max;
      }else{
         right[i]=height[i];
         max = height[i];
      }
   }

   //calculate totoal
   for(int i=0; i<height.length; i++){
      result+= Math.min(left[i],right[i])-height[i];
   }

   return result;
}
```

7 Product of Array Except Self

Given an array of n integers where n >1, nums, return an array output such that output[i] is equal to the product of all the elements of nums except nums[i].

Solve it without division and in O(n).

For example, given [1,2,3,4], return [24,12,8,6].

7.1 Java Solution 1

```
public int[] productExceptSelf(int[] nums) {
   int[] result = new int[nums.length];

   int[] t1 = new int[nums.length];
   int[] t2 = new int[nums.length];

   t1[0]=1;
   t2[nums.length-1]=1;

   //scan from left to right
   for(int i=0; i<nums.length-1; i++){
      t1[i+1] = nums[i] * t1[i];
   }

   //scan from right to left
   for(int i=nums.length-1; i>0; i--){
      t2[i-1] = t2[i] * nums[i];
   }

   //multiply
   for(int i=0; i<nums.length; i++){
      result[i] = t1[i] * t2[i];
   }

   return result;
}
```

7.2 Java Solution 2

We can directly put the product values into the final result array. This saves the extra space to store the 2 intermediate arrays in Solution 1.

```
public int[] productExceptSelf(int[] nums) {
   int[] result = new int[nums.length];

   result[nums.length-1]=1;
   for(int i=nums.length-2; i>=0; i--){
      result[i]=result[i+1]*nums[i+1];
   }
```

```
    int left=1;
    for(int i=0; i<nums.length; i++){
        result[i]=result[i]*left;
        left = left*nums[i];
    }

    return result;
}
```

8 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 $\geq$ 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.

8.1 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.

8.2 Java Solution - two pointers

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;
         sum = sum+nums[i];
         i++;
      }
   }

   if(exists)
      return result;
   else
      return 0;
```

```
}
```

Similarly, we can also write it in a more readable way.

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

   int i=0;
   int j=0;
   int sum=0;

   int minLen = Integer.MAX_VALUE;

   while(j<nums.length){
      if(sum<s){
         sum += nums[j];
         j++;
      }else{
         minLen = Math.min(minLen, j-i);
         if(i==j-1)
            return 1;

         sum -=nums[i];
         i++;
      }
   }

   while(sum>=s){
      minLen = Math.min(minLen, j-i);

      sum -=nums[i++];
   }

   return minLen==Integer.MAX_VALUE? 0: minLen;
}
```

9 Summary Ranges

Given a sorted integer array without duplicates, return the summary of its ranges for consecutive numbers.
For example, given [0,1,2,4,5,7], return ["0->2","4->5","7"].

9.1 Analysis

When iterating over the array, two values need to be tracked: 1) the first value of a new range and 2) the previous value in the range.

9.2 Java Solution

```
public List<String> summaryRanges(int[] nums) {
   List<String> result = new ArrayList<String>();

   if(nums == null || nums.length==0)
      return result;

   if(nums.length==1){
      result.add(nums[0]+"");
   }

   int pre = nums[0]; // previous element
   int first = pre; // first element of each range

   for(int i=1; i<nums.length; i++){
         if(nums[i]==pre+1){
            if(i==nums.length-1){
               result.add(first+"->"+nums[i]);
            }
         }else{
            if(first == pre){
               result.add(first+"");
            }else{
               result.add(first + "->"+pre);
            }

            if(i==nums.length-1){
               result.add(nums[i]+"");
            }

            first = nums[i];
         }

         pre = nums[i];
   }

   return result;
}
```

10 Missing Ranges

Given a sorted integer array nums, where the range of elements are in the inclusive range [lower, upper], return its missing ranges.

Example:

Input: nums = [0, 1, 3, 50, 75], lower = 0 and upper = 99, Output: ["2", "4->49", "51->74", "76->99"]

10.1 Java Solution

```
public List<String> findMissingRanges(int[] nums, int lower, int upper) {
   List<String> result = new ArrayList<>();
   int start = lower;

   if(lower==Integer.MAX_VALUE){
      return result;
   }

   for(int i=0; i<nums.length; i++){
      //handle duplicates, e.g., [1,1,1] lower=1 upper=1
      if(i<nums.length-1 && nums[i]==nums[i+1]){
         continue;
      }

      if(nums[i] == start){
         start++;
      }else{
         result.add(getRange(start, nums[i]-1));
         if(nums[i]==Integer.MAX_VALUE){
            return result;
         }
         start = nums[i]+1;
      }
   }

   if(start<=upper){
      result.add(getRange(start, upper));
   }

   return result;
}

private String getRange(int n1, int n2) {
   return n1 == n2 ? String.valueOf(n1) : String.format("%d->%d" , n1, n2);
}
```

11 Merge Intervals

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].

11.1 Analysis

The key to solve this problem is defining a Comparator first to sort the arraylist of Intevals.

11.2 Java Solution

```
public List<Interval> merge(List<Interval> intervals) {
   if(intervals == null || intervals.size()<=1){
      return intervals;
   }

   Collections.sort(intervals, Comparator.comparing((Interval itl)->itl.start));

   List<Interval> result = new ArrayList<>();
   Interval t = intervals.get(0);

   for(int i=1; i<intervals.size(); i++){
      Interval c = intervals.get(i);
      if(c.start <= t.end){
         t.end = Math.max(t.end, c.end);
      }else{
         result.add(t);
         t = c;
      }
   }

   result.add(t);

   return result;
}
```

12 Insert Interval

Problem:

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

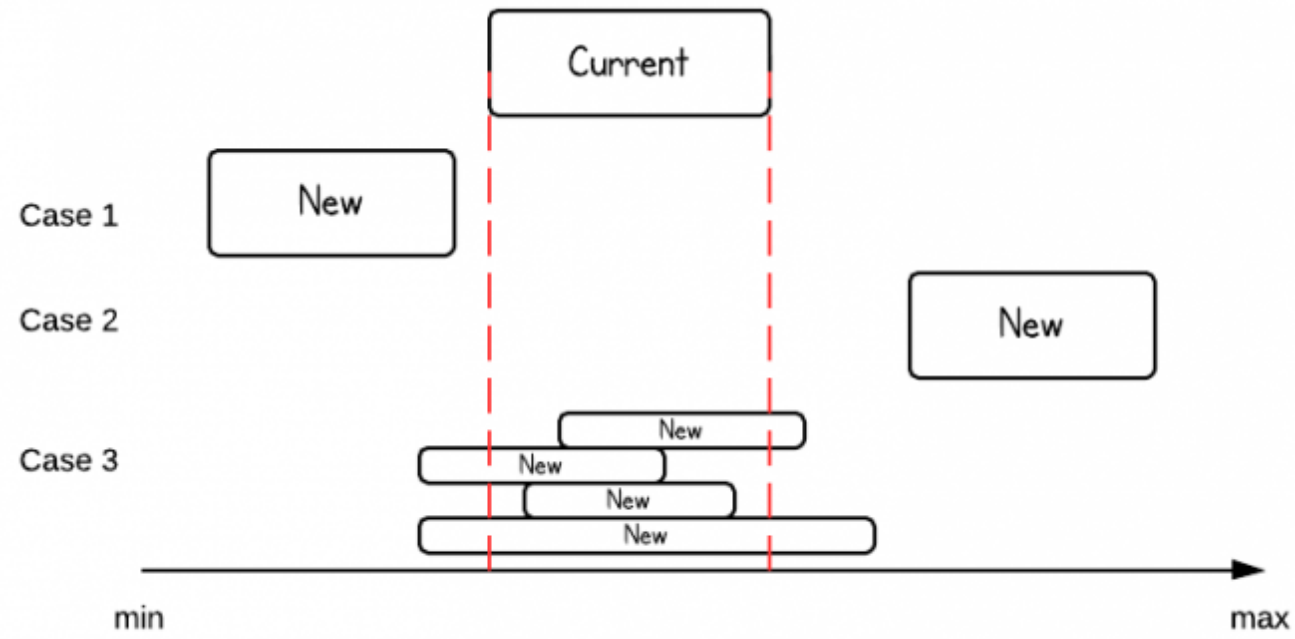
```
Example 1:
Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].

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].
```

12.1 Java Solution 1

When iterating over the list, there are three cases for the current range.




```
/**
 * 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;
   }
}
```

12.2 Java Solution 2 - Binary Search

If the intervals list is an ArrayList, we can use binary search to make the best search time complexity O(log(n)). However, the worst time is bounded by shifting the array list if a new range needs to be inserted. So time complexity is still O(n).

```
public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
   List<Interval> result = new ArrayList<>();

   if (intervals.size() == 0) {
      result.add(newInterval);
      return result;
   }

   int p = helper(intervals, newInterval);
   result.addAll(intervals.subList(0, p));

   for (int i = p; i < intervals.size(); i++) {
      Interval interval = intervals.get(i);
      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;
}

public int helper(List<Interval> intervals, Interval newInterval) {
    int low = 0;
    int high = intervals.size() - 1;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if (newInterval.start <= intervals.get(mid).start) {
            high = mid;
        } else {
            low = mid + 1;
        }
    }

    return high == 0 ? 0 : high - 1;
}
```

The best time is O(log(n)) and worst case time is O(n).

13 Partition Labels

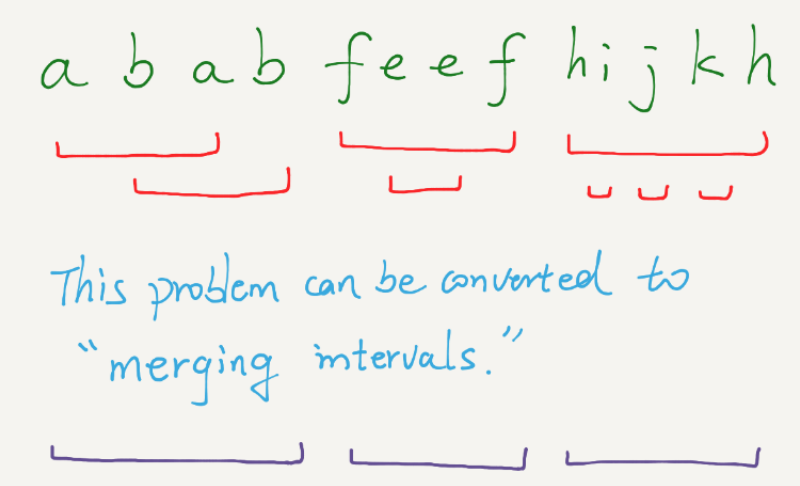
A string S of lowercase letters is given. We want to partition this string into as many parts as possible so that each letter appears in at most one part, and return a list of integers representing the size of these parts.

For example:

Input: S = "ababfeefhijkh" Output: [4,4,5]

Explanation: The partition is "abab", "feef", "hijkh". This is a partition so that each letter appears in at most one part.

13.1 Java Solution




```
public List<Integer> partitionLabels(String S) {
   ArrayList<Integer> result = new ArrayList<>();

   HashMap<Character, int[]> map = new HashMap<>();
   for(int i=0; i<S.length(); i++){
      char c = S.charAt(i);
      int[] arr = map.get(c);

      if(arr == null){
         arr = new int[]{i, i};
         map.put(c, arr);
      }else{
         arr[1]=i;
      }
   }

   ArrayList<int[]> list = new ArrayList<>();
   list.addAll(map.values());

   Collections.sort(list, Comparator.comparing((int[] arr) -> arr[0]));
```

```
   int[] t = list.get(0);
   for(int i=1; i<list.size(); i++){
      int[] range = list.get(i);

      if(range[1]<=t[1]){
         continue;
      }else if(range[0]>t[1]){ //impossible be equal
         result.add(t[1]-t[0]+1);
         t = range;
      }else{
         t[1] = range[1];
      }
   }

   result.add(t[1]-t[0]+1);

   return result;
}
```

14 Find And Replace in String

To some string S, we will perform some replacement operations that replace groups of letters with new ones (not necessarily the same size).

Each replacement operation has 3 parameters: a starting index i, a source word x and a target word y. The rule is that if x starts at position i in the original string S, then we will replace that occurrence of x with y. If not, we do nothing.

For example, if we have S = "abcd" and we have some replacement operation i = 2, x = "cd", y = "ffff", then because "cd" starts at position 2 in the original string S, we will replace it with "ffff".

14.1 Java Solution

```
public String findReplaceString(String S, int[] indexes, String[] sources, String[] targets) {
   StringBuilder sb = new StringBuilder();
   TreeMap<Integer, String[]> map = new TreeMap<>();
   for (int i = 0; i < indexes.length; i++) {
      map.put(indexes[i], new String[]{sources[i], targets[i]});
   }

   int prev = 0;

   for (Map.Entry<Integer, String[]> entry : map.entrySet()) {
      int startIndex = entry.getKey();
      int endIndex = startIndex + entry.getValue()[0].length();

      if (prev != startIndex) {
         sb.append(S.substring(prev, startIndex));
      }

      String org = S.substring(startIndex, endIndex);
      if (org.equals(entry.getValue()[0])) {
         sb.append(entry.getValue()[1]);
         prev = endIndex;
      } else {
         sb.append(org);
         prev = endIndex;
      }

   }

   if (prev < S.length()) {
      sb.append(S.substring(prev));
   }

   return sb.toString();
}
```

15 One Edit Distance

Given two strings S and T, determine if they are both one edit distance apart.

15.1 Java Solution

```
public boolean isOneEditDistance(String s, String t) {
   if(s==null || t==null)
      return false;

   int m = s.length();
   int n = t.length();

   if(Math.abs(m-n)>1){
      return false;
   }

   int i=0;
   int j=0;
   int count=0;

   while(i<m&&j<n){
      if(s.charAt(i)==t.charAt(j)){
         i++;
         j++;
      }else{
         count++;
         if(count>1)
            return false;

         if(m>n){
            i++;
         }else if(m<n){
            j++;
         }else{
            i++;
            j++;
         }
      }
   }

   if(i<m||j<n){
      count++;
   }

   if(count==1)
      return true;

   return false;
}
```

16 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.

16.1 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.

16.2 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--;
      }
   }
}
```

16.3 Java Solution 2

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

```
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--];
  }
}
```

17 Is Subsequence

Given a string s and a string t, check if s is subsequence of t.

You may assume that there is only lower case English letters in both s and t. t is potentially a very long (length = 500,000) string, and s is a short string (<=100).

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, "ace" is a subsequence of "abcde" while "aec" is not).

17.1 Java Solution

```
public boolean isSubsequence(String s, String t) {
   if(s.length()==0)
      return true;

   int i=0;
   int j=0;
   while(i<s.length() && j<t.length()){
      if(s.charAt(i)==t.charAt(j)){
         i++;
      }

      j++;

      if(i==s.length())
         return true;
   }

   return false;
}
```

18 Backspace String Compare

Given two strings S and T, return if they are equal when both are typed into empty text editors. # means a backspace character.

Example 1:

```
Input: S = "ab#c", T = "ad#c"
Output: true
Explanation: Both S and T become "ac".
```

Example 2:

```
Input: S = "a##c", T = "#a#c"
Output: true
Explanation: Both S and T become "c".
```

18.1 Java Solution

This problem requires O(N) time and O(1) space.

```
public boolean backspaceCompare(String S, String T) {
   int i = S.length()-1;
   int j = T.length()-1;

   while(i>=0 || j>=0){
      int c1=0;
      while(i>=0 && (c1>0 || S.charAt(i)=='#')){
         if(S.charAt(i)=='#'){
            c1++;
         }else{
            c1--;
         }

         i--;
      }

      int c2=0;
      while(j>=0 && (c2>0 || T.charAt(j)=='#')){
         if(T.charAt(j)=='#'){
            c2++;
         }else{
            c2--;
         }

         j--;
      }

      if(i>=0 && j>=0){
         if(S.charAt(i)!=T.charAt(j)){
            return false;
```

```
            }else{
                i--;
                j--;
            }
        }else{
            if(i>=0 || j>=0){
                return false;
            }
        }
    }

    return i<0 && j<0;
}
```

19 Repeated String Match

Given two strings A and B, find the minimum number of times A has to be repeated such that B is a substring of it. If no such solution, return -1.

For example, with A = "abcd" and B = "cdabcdab".

Return 3, because by repeating A three times (“abcdabcdabcd”), B is a substring of it; and B is not a substring of A repeated two times ("abcdabcd").

Note: The length of A and B will be between 1 and 10000.

19.1 Java Solution

The optimal solution’s time complexity is O(n) where n is the length of the longer string from A and B.

```
public int repeatedStringMatch(String A, String B) {
   int i = 0;
   int j = 0;

   int result = 0;
   int k = 0;

   while (j < B.length()) {
      if (A.charAt(i) == B.charAt(j)) {
         i++;
         j++;

         if (i == A.length()) {
            i = 0;
            result++;
         }
      } else {
         k++;
         if (k == A.length()) {
            return -1;
         }
         i = k;
         j = 0;
         result = 0;
      }
   }

   if (i > 0) {
      result++;
   }

   return result;
}
```

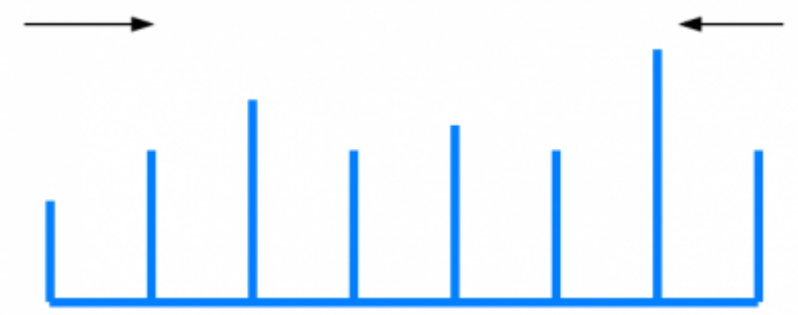
20 Container With Most Water

20.1 Problem

Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of line i is at (i, ai) and (i, 0). Find two lines, which together with x-axis forms a container, such that the container contains the most water.

20.2 Analysis

Initially we can assume the result is 0. Then we scan from both sides. If leftHeight <rightHeight, move right and find a value that is greater than leftHeight. Similarily, if leftHeight >rightHeight, move left and find a value that is greater than rightHeight. Additionally, keep tracking the max value.



20.3 Java Solution

```
public int maxArea(int[] height) {
  if (height == null || height.length < 2) {
    return 0;
  }

  int max = 0;
  int left = 0;
  int right = height.length - 1;

  while (left < right) {
    max = Math.max(max, (right - left) * Math.min(height[left], height[right]));
    if (height[left] < height[right])
      left++;
    else
      right--;
  }

  return max;
}
```

21 Reverse Vowels of a String

Write a function that takes a string as input and reverse only the vowels of a string.

21.1 Java Solution

this is a simple problem which can be solved by using two pointers scanning from beginning and end of the array.

```
public String reverseVowels(String s) {
   ArrayList<Character> vowList = new ArrayList<Character>();
   vowList.add('a');
   vowList.add('e');
   vowList.add('i');
   vowList.add('o');
   vowList.add('u');
   vowList.add('A');
   vowList.add('E');
   vowList.add('I');
   vowList.add('O');
   vowList.add('U');

   char[] arr = s.toCharArray();

   int i=0;
   int j=s.length()-1;

   while(i<j){
      if(!vowList.contains(arr[i])){
         i++;
         continue;
      }

      if(!vowList.contains(arr[j])){
         j--;
         continue;
      }

      char t = arr[i];
      arr[i]=arr[j];
      arr[j]=t;

      i++;
      j--;
   }

   return new String(arr);
}
```

22 Valid Palindrome

Given a string, determine if it is a palindrome, considering only alphanumeric characters 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.

22.1 Java Solution

There are several different ways to solve this problem. The following is a solution with O(n) time complexity and O(1) space complexity.

```
public boolean isPalindrome(String s) {
   if(s==null){
      return false;
   }

   s = s.toLowerCase();

   int i=0;
   int j=s.length()-1;

   while(i<j){
      while(i<j && !((s.charAt(i)>='a' && s.charAt(i)<='z')
               || (s.charAt(i)>='0'&&s.charAt(i)<='9'))){
         i++;
      }

      while(i<j && !((s.charAt(j)>='a' && s.charAt(j)<='z')
               || (s.charAt(j)>='0'&&s.charAt(j)<='9'))){
         j--;
      }

      if(s.charAt(i) != s.charAt(j)){
         return false;
      }

      i++;
      j--;
   }

   return true;
}
```

23 Shortest Word Distance

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = “coding”, word2 = “practice”, return 3. Given word1 = "makes", word2 = "coding", return 1.

23.1 Java Solution

```
public int shortestDistance(String[] words, String word1, String word2) {
  int m=-1;
  int n=-1;

  int min = Integer.MAX_VALUE;

  for(int i=0; i<words.length; i++){
      String s = words[i];
      if(word1.equals(s)){
         m = i;
         if(n!=-1)
            min = Math.min(min, m-n);
      }else if(word2.equals(s)){
         n = i;
         if(m!=-1)
            min = Math.min(min, n-m);
      }
  }

  return min;
}
```

24 Shortest Word Distance II

This is a follow up of Shortest Word Distance. The only difference is now you are given the list of words and your method will be called repeatedly many times with different parameters. How would you optimize it?

Design a class which receives a list of words in the constructor, and implements a method that takes two words word1 and word2 and return the shortest distance between these two words in the list.

For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = “coding”, word2 = “practice”, return 3. Given word1 = "makes", word2 = "coding", return 1.

24.1 Java Solution

```
public class WordDistance {
   HashMap<String, ArrayList<Integer>> map;
   public WordDistance(String[] words) {
      map = new HashMap<String, ArrayList<Integer>>();
      for(int i=0; i<words.length; i++){
         if(map.containsKey(words[i])){
            map.get(words[i]).add(i);
         }else{
            ArrayList<Integer> list = new ArrayList<Integer>();
            list.add(i);
            map.put(words[i], list);
         }
      }
   }

   public int shortest(String word1, String word2) {

      ArrayList<Integer> l1 = map.get(word1);
      ArrayList<Integer> l2 = map.get(word2);

      int result = Integer.MAX_VALUE;
      for(int i1: l1){
         for(int i2: l2){
            result = Math.min(result, Math.abs(i1-i2));
         }
      }
      return result;
   }
}
```

The time complexity for shortest method is O(M*N), where M is freqency of word1 and N is the frequency of word2. This can be improved by the following:

```
public int shortest(String word1, String word2) {

   ArrayList<Integer> l1 = map.get(word1);
   ArrayList<Integer> l2 = map.get(word2);

   int result = Integer.MAX_VALUE;
```

```
   int i=0;
   int j=0;
   while(i<l1.size() && j<l2.size()){
      result = Math.min(result, Math.abs(l1.get(i)-l2.get(j)));
      if(l1.get(i)<l2.get(j)){
         i++;
      }else{
         j++;
      }
   }

   return result;
}
```

The time complexity of the shortest method is now O(M+N). Since M+N <size of word list, the time is O(K) where k is the list size.

25 Shortest Word Distance III

This is a follow-up problem of Shortest Word Distance. The only difference is now word1 could be the same as word2.

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

word1 and word2 may be the same and they represent two individual words in the list.

For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word1 = “makes”, word2 = “coding”, return 1. Given word1 = "makes", word2 = "makes", return 3.

25.1 Java Solution 1

In this problem, word1 and word2 can be the same. The two variables used to track indices should take turns to update.

```
public int shortestWordDistance(String[] words, String word1, String word2) {
  if(words==null || words.length<1 || word1==null || word2==null)
      return 0;

  int m=-1;
  int n=-1;

  int min = Integer.MAX_VALUE;
  int turn=0;
  if(word1.equals(word2))
      turn = 1;

  for(int i=0; i<words.length; i++){
      String s = words[i];
      if(word1.equals(s) && (turn ==1 || turn==0)){
         m = i;
         if(turn==1) turn=2;
         if(n!=-1)
            min = Math.min(min, m-n);
      }else if(word2.equals(s) && (turn==2 || turn==0)){
         n = i;
         if(turn==2) turn =1;
         if(m!=-1)
            min = Math.min(min, n-m);
      }
  }

  return min;
}
```

25.2 Java Solution 2

We can divide the cases to two: word1 and word2 are the same and not the same.