Contents

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?

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).

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) {
     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).

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. Rotate first part: 4,3,2,1,5,6
3. Rotate second part: 4,3,2,1,6,5
4. Rotate the whole array: 5,6,1,2,3,4
```

```
public static void rotate(int[] arr, int order) {
  order = order % arr.length;

  if (arr == null || order < 0) {
    throw new IllegalArgumentException("Illegal argument!");
  }

  //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--;
```

```
    }
}
```

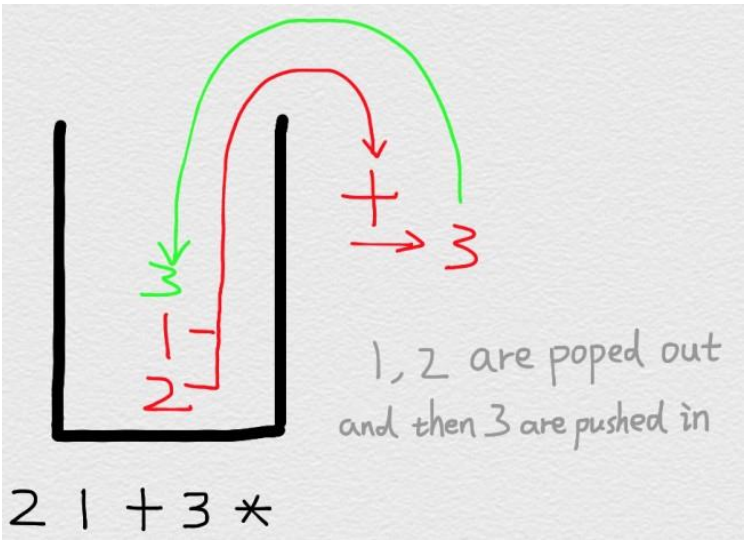
2 Evaluate Reverse Polish Notation

The problem:

```
Evaluate the value of an arithmetic expression in Reverse Polish Notation.

Valid operators are +, -, *, /. Each operand may be an integer or another
    expression.

Some examples:
  ["2", "1", "+", "3", "*"] -> ((2 + 1) * 3) -> 9
  ["4", "13", "5", "/", "+"] -> (4 + (13 / 5)) -> 6
```

Naive Approach

This problem is simple. After understanding the problem, we should quickly realize that 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. It runs great by feeding a small test. However, this code

contains compilation errors in leetcode. Why?

```
public class Test {

  public static void main(String[] args) throws IOException {
    String[] tokens = new String[] { "2", "1", "+", "3", "*" };
    System.out.println(evalRPN(tokens));
  }

  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)) {
        stack.push(t);
      } else {
        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 versions below that.

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;

      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;

   }
}
```

3 Solution of Longest Palindromic Substring in Java

Finding the longest palindromic substring is a classic problem of coding interview. In this post, I will summarize 3 different solutions for this problem.

Naive Approach

Naively, we can simply examine every substring and check if it is palindromic. The time complexity is O(n3). If this is submitted to LeetCode onlinejudge, an error message will be returned - "Time Limit Exceeded". Therefore, this approach is just a start, we need a better algorithm.

```
public static String longestPalindrome1(String s) {

  int maxPalinLength = 0;
  String longestPalindrome = null;
  int length = s.length();

  // check all possible sub strings
  for (int i = 0; i < length; i++) {
    for (int j = i + 1; j < length; j++) {
      int len = j - i;
      String curr = s.substring(i, j + 1);
      if (isPalindrome(curr)) {
        if (len > maxPalinLength) {
          longestPalindrome = curr;
          maxPalinLength = len;
        }
      }
    }
  }

  return longestPalindrome;
}

public static boolean isPalindrome(String s) {

  for (int i = 0; i < s.length() - 1; i++) {
    if (s.charAt(i) != s.charAt(s.length() - 1 - i)) {
      return false;
    }
  }

  return true;
}
```

Dynamic Programming

Let s be the input string, i and j are two indices of the string.

Define a 2-dimension array "table" and let table[i][j] denote whether substring from i to j is palindrome.

Start condition:

```
table[i][i] == 1;
table[i][i+1] == 1 => s.charAt(i) == s.charAt(i+1)
```

Changing condition:

```
table[i+1][j-1] == 1 && s.charAt(i) == s.charAt(j)
=>
table[i][j] == 1
```

Time O(n2) Space O(n2)

```
public static String longestPalindrome2(String s) {
  if (s == null)
    return null;

  if(s.length() <=1)
    return s;

  int maxLen = 0;
  String longestStr = null;

  int length = s.length();

  int[][] table = new int[length][length];

  //every single letter is palindrome
  for (int i = 0; i < length; i++) {
    table[i][i] = 1;
  }
  printTable(table);

  //e.g. bcba
  //two consecutive same letters are palindrome
  for (int i = 0; i <= length - 2; i++) {
    if (s.charAt(i) == s.charAt(i + 1)){
      table[i][i + 1] = 1;
      longestStr = s.substring(i, i + 2);
    }
  }
  printTable(table);
  //condition for calculate whole table
  for (int l = 3; l <= length; l++) {
    for (int i = 0; i <= length-l; i++) {
```

```
        int j = i + l - 1;
        if (s.charAt(i) == s.charAt(j)) {
          table[i][j] = table[i + 1][j - 1];
          if (table[i][j] == 1 && l > maxLen)
            longestStr = s.substring(i, j + 1);
        } else {
          table[i][j] = 0;
        }
        printTable(table);
      }
    }

    return  longestStr;
}
public static void printTable(int[][] x){
    for(int [] y : x){
      for(int z: y){
        System.out.print(z + " ");
      }
      System.out.println();
    }
    System.out.println("----------");
}
```

Given an input, we can use printTable method to examine the table after each iteration. For example, if input string is "dabcba", the final matrix would be the following:

```
1 0 0 0 0 0
0 1 0 0 0 1
0 0 1 0 1 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
```

From the table, we can clear see that the longest string is in cell table[1][5].

Simple Algorithm

From Yifan's comment below.
Time O(n2), Space O(1)

```
public String longestPalindrome(String s) {
    if (s.isEmpty()) {
      return null;
    }

    if (s.length() == 1) {
      return s;
    }
```

```
  String longest = s.substring(0, 1);
  for (int i = 0; i < s.length(); i++) {
    // get longest palindrome with center of i
    String tmp = helper(s, i, i);
    if (tmp.length() > longest.length()) {
      longest = tmp;
    }

    // get longest palindrome with center of i, i+1
    tmp = helper(s, i, i + 1);
    if (tmp.length() > longest.length()) {
      longest = tmp;
    }
  }

  return longest;
}

// Given a center, either one letter or two letter,
// Find longest palindrome
public String helper(String s, int begin, int end) {
  while (begin >= 0 && end <= s.length() - 1 && s.charAt(begin) ==
      s.charAt(end)) {
    begin--;
    end++;
  }
  return s.substring(begin + 1, end);
}
```

Manacher's Algorithm

Manacher's algorithm is much more complicated to figure out, even though it will bring benefit of time complexity of O(n).

Since it is not typical, there is no need to waste time on that.

4 Solution Word Break

Given a string s and a dictionary of words dict, determine if s can be segmented into a space-separated sequence of one or more dictionary words. For example, given s = "leetcode", dict = ["leet", "code"]. Return true because "leetcode" can be segmented as "leet code".

Naive Approach

This problem can be solve by using a naive approach, which is trivial. A discussion can always start from that though.

```
public class Solution {
   public boolean wordBreak(String s, Set<String> dict) {
          return wordBreakHelper(s, dict, 0);
   }

   public boolean wordBreakHelper(String s, Set<String> dict, int start){
      if(start == s.length())
         return true;

      for(String a:  dict){
         int len = a.length();
         int end = start+len;

         //end index should be <= string length
         if(end > s.length())
            continue;

         if(s.substring(start, start+len).equals(a))
            if(wordBreakHelper(s, dict, start+len))
               return true;
      }

      return false;
   }
}
```

Time: O(n2)
This solution exceeds the time limit.

Dynamic Programming

The key to solve this problem by using dynamic programming approach:

- Define an array t[] such that t[i]==true =>0-(i-1) can be segmented using dictionary
- Initial state t[0] == true

```
public class Solution {
   public boolean wordBreak(String s, Set<String> dict) {
      boolean[] t = new boolean[s.length()+1];
      t[0] = true; //set first to be true, why?
      //Because we need initial state

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

```
            //should continue from match position
            if(!t[i])
               continue;

            for(String a:  dict){
               int len = a.length();
               int end = i + len;
               if(end > s.length())
                  continue;

               if(t[end])  continue;

               if(s.substring(i, end).equals(a)){
                  t[end] = true;
               }
            }
         }

         return t[s.length()];
      }
}
```

Time: O(string length * dict size)
One tricky part of this solution is the case:

```
INPUT:  "programcreek",  ["programcree","program","creek"].
```

We should get all possible matches, not stop at "programcree".

Regular Expression

The problem is supposed to be equivalent to matching the regexp (leet|code)*, which means that it can be solved by building a DFA in O(2m̂) and executing it in O(n). (Thanks to hdante.) Leetcode online judge does not allow using Pattern class though.

```
public static void main(String[] args) {
  HashSet<String> dict = new HashSet<String>();
  dict.add("go");
  dict.add("goal");
  dict.add("goals");
  dict.add("special");

  StringBuilder sb = new StringBuilder();

  for(String s: dict){
    sb.append(s + "|");
  }

  String pattern = sb.toString().substring(0, sb.length()-1);
```

```
    pattern = "("+pattern+")*";
    Pattern p = Pattern.compile(pattern);
    Matcher m = p.matcher("goalspecial");

    if(m.matches()){
      System.out.println("match");
    }
}
```

The More Interesting Problem

The dynamic solution can tell us whether the string can be broken to words, but can not tell us what words the string is broken to. So how to get those words?

Check out Word Break II.

5 Word Break II

Given a string s and a dictionary of words dict, add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences.

For example, given s = "catsanddog", dict = ["cat", "cats", "and", "sand", "dog"], the solution is ["cats and dog", "cat sand dog"].

Java Solution - Dynamic Programming

This problem is very similar to Word Break. Instead of using a boolean array to track the match positions, we need to track the actual words. Then we can use depth first search to get all the possible paths, i.e., the list of strings.

The following diagram shows the structure of the tracking array.

	Index	Words
c	0	
a	1	
t	2	
s	3	cat
a	4	cats
n	5	
d	6	
d	7	and, sand
o	8	
g	9	
	10	dog

```
public static List<String> wordBreak(String s, Set<String> dict) {
   //create an array of ArrayList<String>
   List<String> dp[] = new ArrayList[s.length()+1];
   dp[0] = new ArrayList<String>();

   for(int i=0; i<s.length(); i++){
      if( dp[i] == null )
         continue;

      for(String word:dict){
         int len = word.length();
         int end = i+len;
         if(end > s.length())
            continue;

         if(s.substring(i,end).equals(word)){
            if(dp[end] == null){
               dp[end] = new ArrayList<String>();
            }
            dp[end].add(word);
         }
      }
   }

   List<String> result = new LinkedList<String>();
   if(dp[s.length()] == null)
      return result;
```

```
    ArrayList<String> temp = new ArrayList<String>();
    dfs(dp, s.length(), result, temp);

    return result;
}

public static void dfs(List<String> dp[],int end,List<String> result,
     ArrayList<String> tmp){
    if(end <= 0){
        String path = tmp.get(tmp.size()-1);
        for(int i=tmp.size()-2; i>=0; i--){
            path += " " + tmp.get(i) ;
        }

        result.add(path);
        return;
    }

    for(String str : dp[end]){
        tmp.add(str);
        dfs(dp, end-str.length(), result, tmp);
        tmp.remove(tmp.size()-1);
    }
}
```

6 Word Ladder

The problem:

Given two words (start and end), and a dictionary, find the length of shortest transformation sequence from start to end, such that:

Only one letter can be changed at a time Each intermediate word must exist in the dictionary For example,

Given:

```
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
```

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

Note: Return 0 if there is no such transformation sequence. All words have the same length. All words contain only lowercase alphabetic characters.

This problem is a classic problem that has been asked frequently during interviews.

The following are two Java solutions.

Naive Approach

In a simplest way, we can start from start word, change one character each time, if it is in the dictionary, we continue with the replaced word, until start == end.

```
public class Solution {
   public int ladderLength(String start, String end, HashSet<String> dict) {

      int len=0;
      HashSet<String> visited = new HashSet<String>();

      for(int i=0; i<start.length(); i++){
         char[] startArr = start.toCharArray();

         for(char c='a'; c<='z'; c++){
            if(c==start.toCharArray()[i]){
               continue;
            }

            startArr[i] = c;
            String temp = new String(startArr);
            if(dict.contains(temp)){
               len++;
               start = temp;
               if(temp.equals(end)){
                  return len;
               }
            }
         }
      }

      return len;
   }
}
```

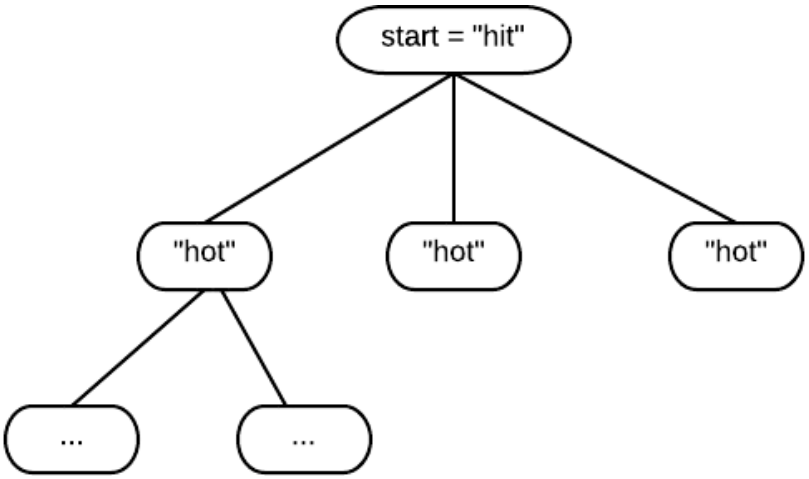
Apparently, this is not good enough. The following example exactly shows the problem. It can not find optimal path. The output is 3, but it actually only takes 2.

```
Input: "a", "c", ["a","b","c"]
Output: 3
Expected: 2
```

Breath First Search

So we quickly realize that this looks like a tree searching problem for which breath first guarantees the optimal solution.

Assuming we have some words in the dictionary, and the start is "hit" as shown in the diagram below.



We can use two queues to traverse the tree, one stores the nodes, the other stores the step numbers.

Updated on 2/27/2015.

```
public int ladderLength(String start, String end, HashSet<String> dict) {
  if (dict.size() == 0)
    return 0;

  dict.add(end);

  LinkedList<String> wordQueue = new LinkedList<String>();
  LinkedList<Integer> distanceQueue = new LinkedList<Integer>();

  wordQueue.add(start);
  distanceQueue.add(1);

  //track the shortest path
  int result = Integer.MAX_VALUE;
  while (!wordQueue.isEmpty()) {
    String currWord = wordQueue.pop();
    Integer currDistance = distanceQueue.pop();

    if (currWord.equals(end)) {
      result = Math.min(result, currDistance);
    }

    for (int i = 0; i < currWord.length(); i++) {
      char[] currCharArr = currWord.toCharArray();
      for (char c = 'a'; c <= 'z'; c++) {
```

```
            currCharArr[i] = c;

            String newWord = new String(currCharArr);
            if (dict.contains(newWord)) {
              wordQueue.add(newWord);
              distanceQueue.add(currDistance + 1);
              dict.remove(newWord);
            }
          }
        }
      }

      if (result < Integer.MAX_VALUE)
        return result;
      else
        return 0;
    }
```

What learned from this problem?

- Use breath-first or depth-first search to solve problems
- Use two queues, one for words and another for counting

7 Median of Two Sorted Arrays Java

LeetCode Problem:

> *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)).*

Java Solution

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.

```
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

    // 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);
  }
```

The Steps of the Algorithm

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

8 Regular Expression Matching in Java

Problem:

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
```

Analysis

First of all, this is one of the most difficulty problems. It is hard to handle many 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 left 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 left.

Be careful about the offset.

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{
         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;
      }
   }
}
```

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) != '.')) {
      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;
  }
}
```

9 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].
```

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/array.

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
    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;
  }
}
```

10 Insert Interval

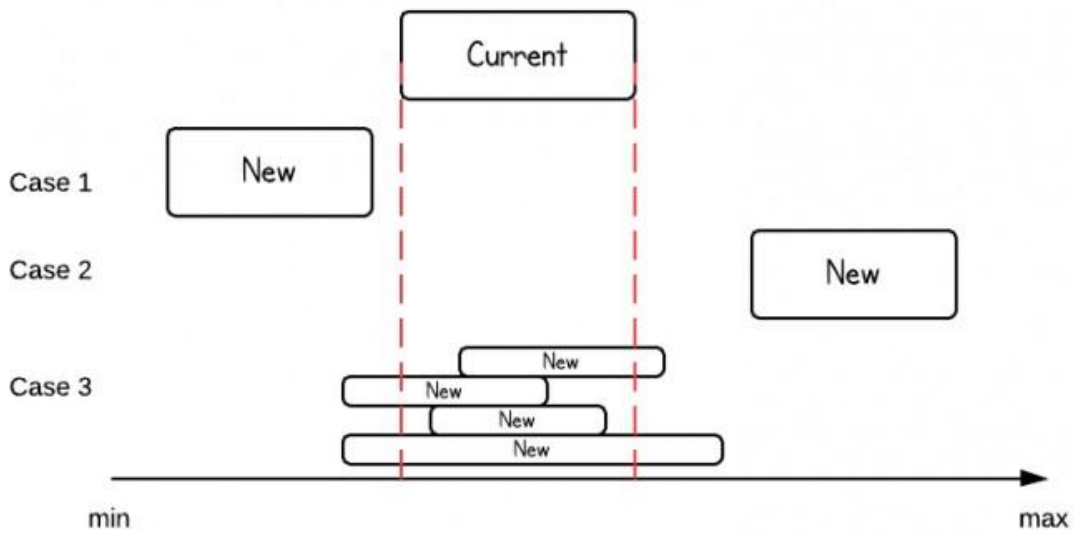
Problem:

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].
```

Thoughts of This Problem

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



Java Solution

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

11 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
```

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(n2).

```
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?

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];

  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 normally O(1).

Time complexity of this solution: O(n).

12 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;
}
```

13 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
```

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;
  }
}
```

14 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 $\leq$ b $\leq$ 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)
```

Naive Solution

Naive solution is 3 loops, and this gives time complexity O(n3). 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();
              }
            }
         }
      }

      return result;
   }
}
```

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

Result:

```
Submission Result: Output Limit Exceeded
```

Better Solution

A better solution is using two pointers instead of one. This makes time complexity of O(n2).

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);
            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;
}
```

15 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)
```

Thoughts

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

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]);
          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());
}
```

16 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).

Thoughts

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

Java Solution

```
public class Solution {
  public int threeSumClosest(int[] num, int target) {
    int min = Integer.MAX_VALUE;
    int result = 0;

    Arrays.sort(num);

    for (int i = 0; i < num.length; i++) {
      int j = i + 1;
```

```
        int k = num.length - 1;
        while (j < k) {
          int sum = num[i] + num[j] + num[k];
          int diff = Math.abs(sum - target);

          if(diff == 0) return 0;

          if (diff < min) {
            min = diff;
            result = sum;
          }
          if (sum <= target) {
            j++;
          } else {
            k--;
          }
        }
      }

      return result;
    }
}
```

Time Complexity is O(n2).

17 String to Integer (atoi)

Problem:

Implement atoi to convert a string to an integer.

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

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

Thoughts for This Problem

The vague description give us space to consider different cases.

```
1. null or empty string
2. white spaces
3. +/- sign
4. calculate real value
5. handle min & max
```

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');
    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;
}
```

Thanks to the comment below. The solution above passes LeetCode online judge, but it haven't considered other characters. I will update this later.

18 Merge Sorted Array

Problem:

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.

Thoughts for This Problem

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.

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--;
      }
   }
}
```

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--];
  }
}
```

19 Valid Parentheses

Problem:

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.

Thoughts about This Problem

Character is not a frequently used class, so need to know how to use it.

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();
}
```

Simplified Java Solution

Almost identical, but convert string to char array at the beginning.

```
public static boolean isValid(String s) {
    char[] charArray = s.toCharArray();

    HashMap<Character, Character> map = new HashMap<Character, Character>();
    map.put('(', ')');
    map.put('[', ']');
    map.put('{', '}');

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

    for (Character c : charArray) {
        if (map.keySet().contains(c)) {
            stack.push(c);
        } else if (map.values().contains(c)) {
            if (!stack.isEmpty() && map.get(stack.peek()) == c) {
                stack.pop();
            } else {
                return false;
            }
        }
    }
    return stack.isEmpty();
}
```

20 Implement strStr()

Problem:

> *Implement strStr(). Returns a pointer to the first occurrence of needle in haystack, or null if needle is not part of haystack.*

Thoughts

First, need to understand the problem correctly, the pointer simply means a sub string. Second, make sure the loop does not exceed the boundaries of two strings.

Java Solution

```
public String strStr(String haystack, String needle) {

  int needleLen = needle.length();
  int haystackLen = haystack.length();

  if (needleLen == haystackLen && needleLen == 0)
    return "";

  if (needleLen == 0)
    return haystack;

  for (int i = 0; i < haystackLen; i++) {
    // make sure in boundary of needle
    if (haystackLen - i + 1 < needleLen)
      return null;

    int k = i;
    int j = 0;

    while (j < needleLen && k < haystackLen && needle.charAt(j) ==
        haystack.charAt(k)) {
      j++;
      k++;
      if (j == needleLen)
        return  haystack.substring(i);
    }

  }

  return null;
}
```

From Tia:

> *You have to check if a String == null before call length(), otherwise it will throw NullPointerException.*

21 Set Matrix Zeroes

Given a m x n matrix, if an element is 0, set its entire row and column to 0. Do it in place.

Thoughts about This Problem

This problem can solve by following 4 steps:

- check if first row and column are zero or not
- mark zeros on first row and column
- use mark to set elements
- set first column and row by using marks in step 1

Java Solution

```
public class Solution {
   public void setZeroes(int[][] matrix) {
      boolean firstRowZero = false;
      boolean firstColumnZero = false;

      //set first row and column zero or not
      for(int i=0; i<matrix.length; i++){
         if(matrix[i][0] == 0){
            firstColumnZero = true;
            break;
         }
      }

      for(int i=0; i<matrix[0].length; i++){
         if(matrix[0][i] == 0){
            firstRowZero = true;
            break;
         }
      }

      //mark zeros on first row and column
      for(int i=1; i<matrix.length; i++){
         for(int j=1; j<matrix[0].length; j++){
            if(matrix[i][j] == 0){
              matrix[i][0] = 0;
              matrix[0][j] = 0;
            }
         }
      }
```

```
        //use mark to set elements
        for(int i=1; i<matrix.length; i++){
            for(int j=1; j<matrix[0].length; j++){
                if(matrix[i][0] == 0 || matrix[0][j] == 0){
                    matrix[i][j] = 0;
                }
            }
        }

        //set first column and row
        if(firstColumnZero){
            for(int i=0; i<matrix.length; i++)
                matrix[i][0] = 0;
        }

        if(firstRowZero){
            for(int i=0; i<matrix[0].length; i++)
                matrix[0][i] = 0;
        }

    }
}
```

22 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
```

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;
    }
}
```

Solution 2

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

```
public class Solution {
    public int searchInsert(int[] A, int target) {
        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);
    }
}
```

23 Longest Consecutive Sequence Java

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.

Thoughts

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

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--;
    }

    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 sequences.

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.

We do have an extreme case here: If n is number of elements, m is average length of consecutive sequence, and m==n, then the time complexity is O(n2). The reason is that in this case, no element is removed from the set each time. You can use this array to get the point: 1,3,5,7,9.

24 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.

Thoughts

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

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--;
                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;
        }
    }
}
```

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;
}
```

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);

    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));
  }
}
```

25 Spiral Matrix

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.

For example, given the following matrix:

```
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
```

You should return [1,2,3,6,9,8,7,4,5].

Java Solution 1

If more than one row and column left, it can form a circle and we process the circle. Otherwise, if only one row or column left, we process that column or row ONLY.

```
public class Solution {
   public ArrayList<Integer> spiralOrder(int[][] matrix) {
      ArrayList<Integer> result = new ArrayList<Integer>();

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

      int m = matrix.length;
      int n = matrix[0].length;

      int x=0;
      int y=0;

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

         //if one row/column left, no circle can be formed
         if(m==1){
            for(int i=0; i<n; i++){
               result.add(matrix[x][y++]);
            }
            break;
         }else if(n==1){
            for(int i=0; i<m; i++){
               result.add(matrix[x++][y]);
            }
            break;
         }

         //below, process a circle

         //top - move right
         for(int i=0;i<n-1;i++){
            result.add(matrix[x][y++]);
         }

         //right - move down
         for(int i=0;i<m-1;i++){
            result.add(matrix[x++][y]);
         }

         //bottom - move left
         for(int i=0;i<n-1;i++){
            result.add(matrix[x][y--]);
         }

         //left - move up
```

```
            for(int i=0;i<m-1;i++){
               result.add(matrix[x--][y]);
            }

            x++;
            y++;
            m=m-2;
            n=n-2;
         }

         return result;
      }
}
```

Java Solution 2

We can also recursively solve this problem. The solution's performance is not better than Solution or as clear as Solution 1. Therefore, Solution 1 should be preferred.

```
public class Solution {
   public ArrayList<Integer> spiralOrder(int[][] matrix) {
      if(matrix==null || matrix.length==0)
         return new ArrayList<Integer>();

      return   spiralOrder(matrix,0,0,matrix.length,matrix[0].length);
   }


   public ArrayList<Integer> spiralOrder(int [][] matrix, int x, int y, int
       m, int n){
      ArrayList<Integer> result = new ArrayList<Integer>();

      if(m<=0||n<=0)
         return result;

      //only one element left
      if(m==1&&n==1) {
         result.add(matrix[x][y]);
         return result;
      }

      //top - move right
      for(int i=0;i<n-1;i++){
         result.add(matrix[x][y++]);
      }

      //right - move down
      for(int i=0;i<m-1;i++){
```

```
            result.add(matrix[x++][y]);
        }

        //bottom - move left
        if(m>1){
            for(int i=0;i<n-1;i++){
                result.add(matrix[x][y--]);
            }
        }

        //left - move up
        if(n>1){
            for(int i=0;i<m-1;i++){
                result.add(matrix[x--][y]);
            }
        }

        if(m==1||n==1)
            result.addAll(spiralOrder(matrix, x, y, 1, 1));
        else
            result.addAll(spiralOrder(matrix, x+1, y+1, m-2, n-2));

        return result;
    }
}
```

26 Search a 2D Matrix

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has properties:

1) Integers in each row are sorted from left to right. 2) The first integer of each row is greater than the last integer of the previous row.

For example, consider the following matrix:

```
[
  [1,  3, 5, 7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
```

Given target = 3, return true.

26.1 Java Solution

This is a typical problem of binary search.

You may try to solve this problem by finding the row first and then the column. There is no need to do that. Because of the matrix's special features, the matrix can be considered as a sorted array. Your goal is to find one element in this sorted array by using binary search.

```
public class Solution {
   public boolean searchMatrix(int[][] matrix, int target) {
      if(matrix==null || matrix.length==0 || matrix[0].length==0)
         return false;

      int m = matrix.length;
      int n = matrix[0].length;

      int start = 0;
      int end = m*n-1;

      while(start<=end){
         int  mid=(start+end)/2;
         int midX=mid/n;
         int midY=mid%n;

         if(matrix[midX][midY]==target)
            return true;

         if(matrix[midX][midY]<target){
            start=mid+1;
         }else{
            end=mid-1;
         }
      }

      return false;
   }
}
```

27 Rotate Image

You are given an n x n 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up: Could you do this in-place?

Naive Solution

In the following solution, a new 2-dimension array is created to store the rotated matrix, and the result is assigned to the matrix at the end. This is WRONG! Why?

```
public class Solution {
   public void rotate(int[][] matrix) {
      if(matrix == null || matrix.length==0)
         return ;

      int m = matrix.length;

      int[][] result = new int[m][m];

      for(int i=0; i<m; i++){
         for(int j=0; j<m; j++){
            result[j][m-1-i] = matrix[i][j];
         }
      }

      matrix = result;
   }
}
```

The problem is that Java is pass by value not by refrence! "matrix" is just a reference to a 2-dimension array. If "matrix" is assigned to a new 2-dimension array in the method, the original array does not change. Therefore, there should be another loop to assign each element to the array referenced by "matrix". Check out "Java pass by value."

```
public class Solution {
   public void rotate(int[][] matrix) {
      if(matrix == null || matrix.length==0)
         return ;

      int m = matrix.length;

      int[][] result = new int[m][m];

      for(int i=0; i<m; i++){
         for(int j=0; j<m; j++){
            result[j][m-1-i] = matrix[i][j];
         }
      }

     for(int i=0; i<m; i++){
         for(int j=0; j<m; j++){
            matrix[i][j] = result[i][j];
         }
      }
```

```
    }
}
```

In-place Solution

By using the relation "matrix[i][j] = matrix[n-1-j][i]", we can loop through the matrix.

```
public void rotate(int[][] matrix) {
  int n = matrix.length;
  for (int i = 0; i < n / 2; i++) {
    for (int j = 0; j < Math.ceil(((double) n) / 2.); j++) {
      int temp = matrix[i][j];
      matrix[i][j] = matrix[n-1-j][i];
      matrix[n-1-j][i] = matrix[n-1-i][n-1-j];
      matrix[n-1-i][n-1-j]  =  matrix[j][n-1-i];
      matrix[j][n-1-i] = temp;
    }
  }
}
```

28 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.

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){
          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;
  }
}
```

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];
}
```

29 Distinct Subsequences Total

Given a string S and a string T, count the number of distinct subsequences of T in S.

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).

Here is an example: S = "rabbbit", T = "rabbit"

Return 3.

Thoughts

When you see string problem that is about subsequence or matching, dynamic programming method should come to your mind naturally. The key is to find the changing condition.

Java Solution 1

Let W(i, j) stand for the number of subsequences of S(0, i) in T(0, j). If S.charAt(i) == T.charAt(j), W(i, j) = W(i-1, j-1) + W(i-1,j); Otherwise, W(i, j) = W(i-1,j).

```
public int numDistincts(String S, String T) {
    int[][] table = new int[S.length() + 1][T.length() + 1];

    for (int i = 0; i < S.length(); i++)
```

```
    table[i][0] = 1;

  for (int i = 1; i <= S.length(); i++) {
    for (int j = 1; j <= T.length(); j++) {
      if (S.charAt(i - 1) == T.charAt(j - 1)) {
        table[i][j] += table[i - 1][j] + table[i - 1][j - 1];
      } else {
        table[i][j] += table[i - 1][j];
      }
    }
  }

  return  table[S.length()][T.length()];
}
```

Java Solution 2

Do NOT write something like this, even it can also pass the online judge.

```
public int numDistinct(String S, String T) {
  HashMap<Character, ArrayList<Integer>> map = new HashMap<Character,
      ArrayList<Integer>>();

  for (int i = 0; i < T.length(); i++) {
    if (map.containsKey(T.charAt(i))) {
      map.get(T.charAt(i)).add(i);
    } else {
      ArrayList<Integer> temp = new ArrayList<Integer>();
      temp.add(i);
      map.put(T.charAt(i),  temp);
    }
  }

  int[] result = new int[T.length() + 1];
  result[0] = 1;

  for (int i = 0; i < S.length(); i++) {
    char c = S.charAt(i);

    if (map.containsKey(c)) {
      ArrayList<Integer> temp = map.get(c);
      int[] old = new int[temp.size()];

      for (int j = 0; j < temp.size(); j++)
        old[j] = result[temp.get(j)];

      // the relation
      for (int j = 0; j < temp.size(); j++)
        result[temp.get(j) + 1] = result[temp.get(j) + 1] + old[j];
```

```
    }
  }

  return  result[T.length()];
}
```

30 Maximum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [−2,1, −3,4, −1,2,1, −5,4], the contiguous subarray [4, −1,2,1] has the largest sum = 6.

Wrong Solution

This is a wrong solution, check out the discussion below to see why it is wrong. I put it here just for fun.

```
public class Solution {
  public int maxSubArray(int[] A) {
    int sum = 0;
    int maxSum = Integer.MIN_VALUE;

    for (int i = 0; i < A.length; i++) {
      sum += A[i];
      maxSum = Math.max(maxSum, sum);

      if (sum < 0)
        sum = 0;
    }

    return maxSum;
  }
}
```

Dynamic Programming Solution

The changing condition for dynamic programming is "We should ignore the sum of the previous n-1 elements if nth element is greater than the sum."

```
public class Solution {
  public int maxSubArray(int[] A) {
```

```
    int max = A[0];
    int[] sum = new int[A.length];
    sum[0] = A[0];

    for (int i = 1; i < A.length; i++) {
      sum[i] = Math.max(A[i], sum[i - 1] + A[i]);
      max = Math.max(max, sum[i]);
    }

    return max;
  }
}
```

Simple Solution

Mehdi provided the following solution in his comment.

```
public int maxSubArray(int[] A) {
     int newsum=A[0];
     int max=A[0];
     for(int i=1;i<A.length;i++){
        newsum=Math.max(newsum+A[i],A[i]);
        max= Math.max(max, newsum);
     }
     return max;
   }
```

This problem is asked by Palantir.

31 Maximum Product Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array [2,3,-2,4], the contiguous subarray [2,3] has the largest product = 6.

Java Solution 1 - Brute-force

```
public int maxProduct(int[] A) {
   int max = Integer.MIN_VALUE;

   for(int i=0; i<A.length; i++){
      for(int l=0; l<A.length; l++){
```

```
            if(i+l < A.length){
               int product = calProduct(A, i, l);
               max = Math.max(product, max);
            }

         }

      }
      return max;
}

public int calProduct(int[] A, int i, int j){
      int result = 1;
      for(int m=i; m<=j; m++){
         result = result * A[m];
      }
      return result;
}
```

The time of the solution is O(n3).

Java Solution 2 - Dynamic Programming

This is similar to maximum subarray. Instead of sum, the sign of number affect the product value.

When iterating the array, each element has two possibilities: positive number or negative number. We need to track a minimum value, so that when a negative number is given, it can also find the maximum value. We define two local variables, one tracks the maximum and the other tracks the minimum.

```
public int maxProduct(int[] A) {
   if(A==null || A.length==0)
      return 0;

   int maxLocal = A[0];
   int minLocal = A[0];
   int global = A[0];

   for(int i=1; i<A.length; i++){
      int temp = maxLocal;
      maxLocal = Math.max(Math.max(A[i]*maxLocal, A[i]), A[i]*minLocal);
      minLocal = Math.min(Math.min(A[i]*temp, A[i]), A[i]*minLocal);
      global = Math.max(global, maxLocal);
   }
   return global;
}
```

Time is O(n).

32 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].

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.

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

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.

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++;
    }
  }
  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);
}
```

33 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].

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;
    }
}
```

Online Judge returns:

```
Submission Result: Wrong Answer
Input: [1,1,1,2]
Output: [1,1,1]
Expected: [1,1,2]
```

So this problem also requires in-place array manipulation.

Correct Solution

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 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;

    // index for updating
    int o = 1;

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

      if (curr == pre) {
        if (!flag) {
          flag = true;
          A[o++] = curr;

          continue;
        } else {
          count++;
        }
      } else {
        pre = curr;
        A[o++] = curr;
        flag = false;
      }
    }
```

```
    return A.length - count;
  }
}
```

Better Solution

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

    int prev = 1; // point to previous
    int curr = 2; // point to current

    while (curr < A.length) {
      if (A[curr] == A[prev] && A[curr] == A[prev - 1]) {
        curr++;
      } else {
        prev++;
        A[prev] = A[curr];
        curr++;
      }
    }

    return prev + 1;
  }
}
```

34 Longest Substring Without Repeating Characters

Given a string, find the length of the longest substring without repeating characters. For example, the longest substring without repeating letters for "abcabcbb" is "abc", which the length is 3. For "bbbbb" the longest substring is "b", with the length of 1.

Java Solution 1

The first solution is like the problem of "determine if a string has all unique characters" in CC 150. We can use a flag array to track the existing characters for the longest substring without repeating characters.

```
public int lengthOfLongestSubstring(String s) {
  boolean[] flag = new boolean[256];

  int result = 0;
  int start = 0;
  char[] arr = s.toCharArray();

  for (int i = 0; i < arr.length; i++) {
    char current = arr[i];
    if (flag[current]) {
      result = Math.max(result, i - start);
      // the loop update the new start point
      // and reset flag array
      // for example, abccab, when it comes to 2nd c,
      // it update start from 0 to 3, reset flag for a,b
      for (int k = start; k < i; k++) {
        if (arr[k] == current) {
          start = k + 1;
          break;
        }
        flag[arr[k]] = false;
      }
    } else {
      flag[current] = true;
    }
  }

  result = Math.max(arr.length - start, result);

  return result;
}
```

Java Solution 2

This solution is from Tia. It is easier to understand than the first solution.

The basic idea is using a hash table to track existing characters and their position. When a repeated character occurs, check from the previously repeated character. However, the time complexity is higher - O(n3).

```
public static int lengthOfLongestSubstring(String s) {

  char[] arr = s.toCharArray();
  int pre = 0;

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

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

```
      if (!map.containsKey(arr[i])) {
        map.put(arr[i], i);
      } else {
        pre = Math.max(pre, map.size());
        i = map.get(arr[i]);
        map.clear();
      }
    }

    return Math.max(pre, map.size());
}
```

Consider the following simple example.

```
abcda
```

When loop hits the second "a", the HashMap contains the following:

```
a 0
b 1
c 2
d 3
```

The index i is set to 0 and incremented by 1, so the loop start from second element again.

35 Longest Substring Which Contains 2 Unique Characters

This is a problem asked by Google.

Problem

Given a string, find the longest substring that contains only two unique characters. For example, given "abcbbbbcccbdddadacb", the longest substring that contains 2 unique character is "bcbbbbcccb".

Naive Solution

Here is a naive solution. It works. Basically, it has two pointers that track the start of the substring and the iteration cursor.

```
public static String subString(String s) {
  // checking
```

```
  char[] arr = s.toCharArray();
  int max = 0;
  int j = 0;
  int m = 0, n = 0;

  HashSet<Character> set = new HashSet<Character>();
  set.add(arr[0]);

  for (int i = 1; i < arr.length; i++) {
    if (set.add(arr[i])) {
      if (set.size() > 2) {
        String str = s.substring(j, i);

        //keep the last character only
        set.clear();
        set.add(arr[i - 1]);

        if ((i - j) > max) {
          m = j;
          n = i - 1;
          max = i - j;
        }

        j = i - helper(str);
      }
    }
  }

  return s.substring(m, n + 1);
}

// This method returns the length that contains only one character from right
    side.
public static int helper(String str) {
  // null & illegal checking here
  if(str == null){
    return 0;
  }

  if(str.length() == 1){
    return 1;
  }

  char[] arr = str.toCharArray();
  char p = arr[arr.length - 1];
  int result = 1;

  for (int i = arr.length - 2; i >= 0; i--) {
    if (p == arr[i]) {
```

```
        result++;
      } else {
        break;
      }
    }

    return result;
}
```

Now if this question is extended to be "the longest substring that contains k unique characters", what should we do? Apparently, the solution above is not scalable.

Scalable Solution

The above solution can be extended to be a more general solution which would allow k distinct characters.

36 Palindrome Partitioning

Problem

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.
For example, given s = "aab", Return

```
[
  ["aa","b"],
  ["a","a","b"]
]
```

Java Solution 1

```
public ArrayList<ArrayList<String>> partition(String s) {
    ArrayList<ArrayList<String>> result = new ArrayList<ArrayList<String>>();

    if (s == null || s.length() == 0) {
      return result;
    }

    ArrayList<String> partition = new ArrayList<String>();
    addPalindrome(s, 0, partition, result);
```

```
  return result;
}

private void addPalindrome(String s, int start, ArrayList<String> partition,
    ArrayList<ArrayList<String>> result) {
  //stop condition
  if (start == s.length()) {
    ArrayList<String> temp = new ArrayList<String>(partition);
    result.add(temp);
    return;
  }

  for (int i = start + 1; i <= s.length(); i++) {
    String str = s.substring(start, i);
    if (isPalindrome(str)) {
      partition.add(str);
      addPalindrome(s, i, partition, result);
      partition.remove(partition.size() - 1);
    }
  }
}

private boolean isPalindrome(String str) {
  int left = 0;
  int right = str.length() - 1;

  while (left < right) {
    if (str.charAt(left) != str.charAt(right)) {
      return false;
    }

    left++;
    right--;
  }

  return true;
}
```

Dynamic Programming

The dynamic programming approach is very similar to the problem of longest palindrome substring.

```
public static List<String> palindromePartitioning(String s) {

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

  if (s == null)
```

```
    return result;

  if (s.length() <= 1) {
    result.add(s);
    return result;
  }

  int length = s.length();

  int[][] table = new int[length][length];

  // l is length, i is index of left boundary, j is index of right boundary
  for (int l = 1; l <= length; l++) {
    for (int i = 0; i <= length - l; i++) {
      int j = i + l - 1;
      if (s.charAt(i) == s.charAt(j)) {
        if (l == 1 || l == 2) {
          table[i][j] = 1;
        } else {
          table[i][j] = table[i + 1][j - 1];
        }
        if (table[i][j] == 1) {
          result.add(s.substring(i, j + 1));
        }
      } else {
        table[i][j] = 0;
      }
    }
  }

  return result;
}
```

37 Reverse Words in a String

Given an input string, reverse the string word by word.

For example, given s = "the sky is blue", return "blue is sky the".

Java Solution

This problem is pretty straightforward. We first split the string to words array, and then iterate through the array and add each element to a new string. Note: StringBuilder should be used to avoid creating too many Strings. If the string is very long,

using String is not scalable since String is immutable and too many objects will be created and garbage collected.

```
class Solution {
  public String reverseWords(String s) {
    if (s == null || s.length() == 0) {
      return "";
    }

    // split to words by space
    String[] arr = s.split(" ");
    StringBuilder sb = new StringBuilder();
    for (int i = arr.length - 1; i >= 0; --i) {
      if (!arr[i].equals("")) {
        sb.append(arr[i]).append("  ");
      }
    }
    return sb.length() == 0 ? "" : sb.substring(0, sb.length() - 1);
  }
}
```

38 Find Minimum in Rotated Sorted Array

Suppose a sorted array is rotated at some pivot unknown to you beforehand. (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).

Find the minimum element.You may assume no duplicate exists in the array.

Thoughts

When we search something from a sorted array, binary search is almost a top choice. Binary search is efficient for sorted arrays.

This problems seems like a binary search, and the key is how to break the array to two parts, so that we only need to work on half of the array each time, i.e., when to select the left half and when to select the right half.

If we pick the middle element, we can compare the middle element with the left-end element. If middle is less than leftmost, the left half should be selected; if the middle is greater than the leftmost, the right half should be selected. Using simple recursion, this problem can be solve in time log(n).

In addition, in any rotated sorted array, the rightmost element should be less than the left-most element, otherwise, the sorted array is not rotated and we can simply

pick the leftmost element as the minimum.

Java Solution

Define a helper function, otherwise, we will need to use Arrays.copyOfRange() function, which may be expensive for large arrays.

```
public int findMin(int[] num) {
  return findMin(num, 0, num.length - 1);
}

public int findMin(int[] num, int left, int right) {
  if (left == right)
    return num[left];
  if ((right - left) == 1)
    return Math.min(num[left], num[right]);

  int middle = left + (right - left) / 2;

  // not rotated
  if (num[left] < num[right]) {
    return num[left];

  // go right side
  } else if (num[middle] > num[left]) {
    return findMin(num, middle, right);

  // go left side
  } else {
    return findMin(num, left, middle);
  }
}
```

39 Find Minimum in Rotated Sorted Array II

Problem

Follow up for "Find Minimum in Rotated Sorted Array": What if duplicates are allowed?

Would this affect the run-time complexity? How and why?

Java Solution

This is a follow-up problem of finding minimum element in rotated sorted array without duplicate elements. We only need to add one more condition, which checks if the left-most element and the right-most element are equal. If they are we can simply drop one of them. In my solution below, I drop the left element whenever the left-most equals to the right-most.

```
public int findMin(int[] num) {
    return findMin(num, 0, num.length-1);
}

public int findMin(int[] num, int left, int right){
    if(right==left){
        return num[left];
    }
    if(right == left +1){
        return Math.min(num[left], num[right]);
    }
    // 3 3 1 3 3 3

    int middle = (right-left)/2 + left;
    // already sorted
    if(num[right] > num[left]){
        return num[left];
    //right shift one
    }else if(num[right] == num[left]){
        return findMin(num, left+1, right);
    //go right
    }else if(num[middle] >= num[left]){
        return findMin(num, middle, right);
    //go left
    }else{
        return findMin(num, left, middle);
    }
}
```

40 Find Peak Element

A peak element is an element that is greater than its neighbors. Given an input array where num[i] ≠ num[i+1], find a peak element and return its index. The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that num[-1] = num[n] = -∞. For example, in array [1, 2, 3, 1], 3 is

a peak element and your function should return the index number 2.

Thoughts

This is a very simple problem. We can scan the array and find any element that is greater can its previous and next. The first and last element are handled separately.

Java Solution

```
public class Solution {
   public int findPeakElement(int[] num) {
      int max = num[0];
      int index = 0;
      for(int i=1; i<=num.length-2; i++){
         int prev = num[i-1];
         int curr = num[i];
         int next = num[i+1];

         if(curr > prev && curr > next && curr > max){
            index = i;
            max = curr;
         }
      }

      if(num[num.length-1] > max){
         return num.length-1;
      }

      return index;
   }
}
```

41 Min Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

push(x) – Push element x onto stack. pop() – Removes the element on top of the stack. top() – Get the top element. getMin() – Retrieve the minimum element in the stack.

Thoughts

An array is a perfect fit for this problem. We can use a integer to track the top of the stack. You can use the Stack class from Java SDK, but I think a simple array is more efficient and more beautiful.

Java Solution

```
class MinStack {
   private int[] arr = new int[100];
   private int index = -1;

   public void push(int x) {
      if(index == arr.length - 1){
         arr = Arrays.copyOf(arr, arr.length*2);
      }
      arr[++index] = x;
   }

   public void pop() {
      if(index>-1){
         if(index == arr.length/2 && arr.length > 100){
            arr = Arrays.copyOf(arr, arr.length/2);
         }
         index--;
      }
   }

   public int top() {
      if(index > -1){
         return arr[index];
      }else{
         return 0;
      }
   }

   public int getMin() {
      int min = Integer.MAX_VALUE;
      for(int i=0; i<=index; i++){
         if(arr[i] < min)
            min = arr[i];
      }
      return min;
   }
}
```

42 Majority Element

Problem:

Given an array of size n, find the majority element. The majority element is the element that appears more than ⌊n/2⌋ times. You may assume that the array is non-empty and the majority element always exist in the array.

Java Solution 1

We can sort the array first, which takes time of nlog(n). Then scan once to find the longest consecutive substrings.

```
public class Solution {
   public int majorityElement(int[] num) {
      if(num.length==1){
         return num[0];
      }

      Arrays.sort(num);

      int prev=num[0];
      int count=1;
      for(int i=1; i<num.length; i++){
         if(num[i] == prev){
            count++;
            if(count > num.length/2) return num[i];
         }else{
            count=1;
            prev = num[i];
         }
      }

      return 0;
   }
}
```

Java Solution 2 - Much Simpler

Thanks to SK. His/her solution is much efficient and simpler. Since the majority always take more than a half space, the middle element is guaranteed to be the majority. Sorting array takes nlog(n). So the time complexity of this solution is nlog(n). Cheers!

```
public int majorityElement(int[] num) {
  if (num.length == 1) {
    return num[0];
  }
```

```
    Arrays.sort(num);
    return num[num.length / 2];
}
```

43 Combination Sum

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T. The same repeated number may be chosen from C unlimited number of times.

Note: All numbers (including target) will be positive integers. Elements in a combination (a1, a2, ... , ak) must be in non-descending order. (ie, a1 <= a2 <= ... <= ak). The solution set must not contain duplicate combinations. For example, given candidate set 2,3,6,7 and target 7, A solution set is:

```
[7]
[2, 2, 3]
```

Thoughts

The first impression of this problem should be depth-first search(DFS). To solve DFS problem, recursion is a normal implementation.

Note that the candidates array is not sorted, we need to sort it first.

Java Solution

```
public ArrayList<ArrayList<Integer>> combinationSum(int[] candidates, int
    target) {
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

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

    ArrayList<Integer> current = new ArrayList<Integer>();
    Arrays.sort(candidates);

    combinationSum(candidates, target, 0, current, result);

    return result;
}
```

```
public void combinationSum(int[] candidates, int target, int j,
    ArrayList<Integer>  curr,  ArrayList<ArrayList<Integer>>  result){
  if(target == 0){
     ArrayList<Integer> temp = new ArrayList<Integer>(curr);
     result.add(temp);
     return;
  }

  for(int i=j; i<candidates.length; i++){
     if(target < candidates[i])
         return;

     curr.add(candidates[i]);
     combinationSum(candidates, target - candidates[i], i, curr, result);
     curr.remove(curr.size()-1);
  }
}
```

44 Best Time to Buy and Sell Stock

Say you have an array for which the ith element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Naive Approach

The naive approach exceeds time limit.

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

   int profit = Integer.MIN_VALUE;
   for(int i=0; i<prices.length-1; i++){
      for(int j=0; j< prices.length; j++){
         if(profit < prices[j] - prices[i]){
            profit = prices[j] - prices[i];
         }
      }
   }
   return profit;
}
```

Efficient Approach

Instead of keeping track of largest element in the array, we track the maximum profit so far.

```
public int maxProfit(int[] prices) {
   int profit = 0;
   int minElement = Integer.MAX_VALUE;
   for(int i=0; i<prices.length; i++){
     profit = Math.max(profit, prices[i]-minElement);
     minElement = Math.min(minElement, prices[i]);
   }
   return profit;
}
```

45 Best Time to Buy and Sell Stock II

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Analysis

This problem can be viewed as finding all ascending sequences. For example, given 5, 1, 2, 3, 4, buy at 1 & sell at 4 is the same as buy at 1 &sell at 2 & buy at 2& sell at 3 & buy at 3 & sell at 4.

We can scan the array once, and find all pairs of elements that are in ascending order.

Java Solution

```
public int maxProfit(int[] prices) {
   int profit = 0;
   for(int i=1; i<prices.length; i++){
      int diff = prices[i]-prices[i-1];
      if(diff > 0){
         profit += diff;
      }
   }
```

```
    return profit;
}
```

46 Best Time to Buy and Sell Stock III

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note: A transaction is a buy & a sell. You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Analysis

Comparing to I and II, III limits the number of transactions to 2. This can be solve by "devide and conquer". We use left[i] to track the maximum profit for transactions before i, and use right[i] to track the maximum profit for transactions after i. You can use the following example to understand the Java solution:

```
Prices: 1 4 5 7 6 3 2 9
left = [0, 3, 4, 6, 6, 6, 6, 8]
right= [8, 7, 7, 7, 7, 7, 7, 0]
```

The maximum profit = 13

Java Solution

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

  //highest profit in 0 ... i
  int[] left = new int[prices.length];
  int[] right = new int[prices.length];

  // DP from left to right
  left[0] = 0;
  int min = prices[0];
  for (int i = 1; i < prices.length; i++) {
    min = Math.min(min, prices[i]);
    left[i] = Math.max(left[i - 1], prices[i] - min);
```

```
    }

    // DP from right to left
    right[prices.length - 1] = 0;
    int max = prices[prices.length - 1];
    for (int i = prices.length - 2; i >= 0; i--) {
      max = Math.max(max, prices[i]);
      right[i] = Math.max(right[i + 1], max - prices[i]);
    }

    int profit = 0;
    for (int i = 0; i < prices.length; i++) {
      profit = Math.max(profit, left[i] + right[i]);
    }

    return profit;
}
```

47 Best Time to Buy and Sell Stock IV

Problem

Say you have an array for which the ith element is the price of a given stock on day i.Design an algorithm to find the maximum profit. You may complete at most k transactions.

Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Analysis

This is a generalized version of Best Time to Buy and Sell Stock III. If we can solve this problem, we can also use k=2 to solve III.

The problem can be solve by using dynamic programming. The relation is:

```
local[i][j] = max(global[i-1][j-1] + max(diff,0), local[i-1][j]+diff)
global[i][j] = max(local[i][j], global[i-1][j])
```

We track two arrays - local and global. The local array tracks maximum profit of j transactions & the last transaction is on ith day. The global array tracks the maximum profit of j transactions until ith day.

Java Solution - 2D Dynamic Programming

```
public int maxProfit(int k, int[] prices) {
  int len = prices.length;

  if (len < 2 || k <= 0)
    return 0;

  // ignore this line
  if (k == 1000000000)
    return 1648961;

  int[][] local = new int[len][k + 1];
  int[][] global = new int[len][k + 1];

  for (int i = 1; i < len; i++) {
    int diff = prices[i] - prices[i - 1];
    for (int j = 1; j <= k; j++) {
      local[i][j] = Math.max(
          global[i - 1][j - 1] + Math.max(diff, 0),
          local[i - 1][j] + diff);
      global[i][j] = Math.max(global[i - 1][j], local[i][j]);
    }
  }

  return global[prices.length - 1][k];
}
```

Java Solution - 1D Dynamic Programming

The solution above can be simplified to be the following:

```
public int maxProfit(int k, int[] prices) {
  if (prices.length < 2 || k <= 0)
    return 0;

  //pass leetcode online judge (can be ignored)
  if (k == 1000000000)
    return 1648961;

  int[] local = new int[k + 1];
  int[] global = new int[k + 1];

  for (int i = 0; i < prices.length - 1; i++) {
    int diff = prices[i + 1] - prices[i];
    for (int j = k; j >= 1; j--) {
      local[j] = Math.max(global[j - 1] + Math.max(diff, 0), local[j] + diff);
      global[j] = Math.max(local[j], global[j]);
```

```
    }
  }

  return global[k];
}
```

48 Longest Common Prefix

Problem

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

Analysis

To solve this problem, we need to find the two loop conditions. One is the length of the shortest string. The other is iteration over every element of the string array.

Java Solution

```
public String longestCommonPrefix(String[] strs) {
   if(strs == null || strs.length == 0)
      return "";

   int minLen=Integer.MAX_VALUE;
   for(String str: strs){
      if(minLen > str.length())
         minLen = str.length();
   }
   if(minLen == 0) return "";

   for(int j=0; j<minLen; j++){
      char prev='0';
      for(int i=0; i<strs.length ;i++){
         if(i==0) {
            prev = strs[i].charAt(j);
            continue;
         }

         if(strs[i].charAt(j) != prev){
            return strs[i].substring(0, j);
         }
      }
```

```
    }

    return  strs[0].substring(0,minLen);
}
```

49 Largest Number

Problem

Given a list of non negative integers, arrange them such that they form the largest number.

For example, given [3, 30, 34, 5, 9], the largest formed number is 9534330.

Note: The result may be very large, so you need to return a string instead of an integer.

Analysis

This problem can be solve by simply sorting strings, not sorting integer. Define a comparator to compare strings by concat() right-to-left or left-to-right.

Java solution

```
public String largestNumber(int[] num) {
    String[] NUM = new String[num.length];

    for (int i = 0; i <num.length; i++) {
        NUM[i] = String.valueOf(num[i]);
    }

    java.util.Arrays.sort(NUM,  new  java.util.Comparator<String>()  {
        public int compare(String left, String right) {
            String leftRight = left.concat(right);
            String rightLeft = right.concat(left);
            return  rightLeft.compareTo(leftRight);
        }
    });

    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < NUM.length; i++) {
        sb.append(NUM[i]);
    }
```

```
    java.math.BigInteger result = new java.math.BigInteger(sb.toString());
    return result.toString();
}
```

50 Combinations

Problem

Given two integers n and k, return all possible combinations of k numbers out of 1 ... n.

For example, if n = 4 and k = 2, a solution is:

```
[
  [2,4],
  [3,4],
  [2,3],
  [1,2],
  [1,3],
  [1,4],
]
```

Java Solution 1 (Recursion)

This is my naive solution. It passed the online judge. I first initialize a list with only one element, and then recursively add available elements to it.

```
public ArrayList<ArrayList<Integer>> combine(int n, int k) {
  ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

  //illegal case
  if (k > n) {
    return null;
  //if k==n
  } else if (k == n) {
    ArrayList<Integer> temp = new ArrayList<Integer>();
    for (int i = 1; i <= n; i++) {
      temp.add(i);
    }
    result.add(temp);
    return result;
  //if k==1
```

```
  } else if (k == 1) {

    for (int i = 1; i <= n; i++) {
      ArrayList<Integer> temp = new ArrayList<Integer>();
      temp.add(i);
      result.add(temp);
    }

    return result;
  }

  //for normal cases, initialize a list with one element
  for (int i = 1; i <= n - k + 1; i++) {
    ArrayList<Integer> temp = new ArrayList<Integer>();
    temp.add(i);
    result.add(temp);
  }

  //recursively add more elements
  combine(n, k, result);

  return result;
}

public void combine(int n, int k, ArrayList<ArrayList<Integer>> result) {
  ArrayList<ArrayList<Integer>> prevResult = new
      ArrayList<ArrayList<Integer>>();
  prevResult.addAll(result);

  if(result.get(0).size() == k) return;

  result.clear();
  for (ArrayList<Integer> one : prevResult) {

    for (int i = 1; i <= n; i++) {
      if (i > one.get(one.size() - 1)) {
        ArrayList<Integer> temp = new ArrayList<Integer>();
        temp.addAll(one);
        temp.add(i);
        result.add(temp);
      }
    }
  }

  combine(n, k, result);
}
```

Java Solution 2 - DFS

```
public ArrayList<ArrayList<Integer>> combine(int n, int k) {
  ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

  if (n <= 0 || n < k)
    return result;

  ArrayList<Integer> item = new ArrayList<Integer>();
  dfs(n, k, 1, item, result); // because it need to begin from 1

  return result;
}

private void dfs(int n, int k, int start, ArrayList<Integer> item,
    ArrayList<ArrayList<Integer>> res) {
  if (item.size() == k) {
    res.add(new ArrayList<Integer>(item));
    return;
  }

  for (int i = start; i <= n; i++) {
    item.add(i);
    dfs(n, k, i + 1, item, res);
    item.remove(item.size() - 1);
  }
}
```

51 Compare Version Numbers

Problem

Compare two version numbers version1 and version2. If version1 >version2 return 1, if version1 <version2 return -1, otherwise return 0.

You may assume that the version strings are non-empty and contain only digits and the . character. The . character does not represent a decimal point and is used to separate number sequences.

Here is an example of version numbers ordering:

```
0.1 < 1.1 < 1.2 < 13.37
```

Java Solution

The tricky part of the problem is to handle cases like 1.0 and 1. They should be equal.

```
public int compareVersion(String version1, String version2) {
   String[] arr1 = version1.split("\\.");
   String[] arr2 = version2.split("\\.");

   int i=0;
   while(i<arr1.length || i<arr2.length){
      if(i<arr1.length && i<arr2.length){
         if(Integer.parseInt(arr1[i]) < Integer.parseInt(arr2[i])){
            return -1;
         }else  if(Integer.parseInt(arr1[i])  >  Integer.parseInt(arr2[i])){
            return 1;
         }
      } else if(i<arr1.length){
         if(Integer.parseInt(arr1[i]) != 0){
            return 1;
         }
      } else if(i<arr2.length){
        if(Integer.parseInt(arr2[i]) != 0){
            return -1;
         }
      }

      i++;
   }

   return 0;
}
```

52 Gas Station

Problem

There are N gas stations along a circular route, where the amount of gas at station i is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from station i to its next station (i+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

Analysis

To solve this problem, we need to understand: 1) if sum of gas[] >= sum of cost[], then there exists a start index to complete the circle. 2) if A can not read C in a the sequence of A–>B–>C, then B can not make it either.

Proof:

```
If gas[A] < cost[A], then A can not go to B. Therefore, gas[A] >=cost[A].
We already know A can not go to C, we have gas[A] + gas[B] < cost[A] + cost[B]
And gas[A] >=cost[A],
Therefore, gas[B] < cost[B], i.e., B can not go to C.
```

In the following solution, sumRemaining tracks the sum of remaining to the current index. If sumRemaining <0, then every index between old start and current index is bad, and we need to update start to be the current index.

index	0	1	2	3	4
gas	1	2	3	4	5
cost	1	3	2	4	5

Java Solution

```
public int canCompleteCircuit(int[] gas, int[] cost) {
  int sumRemaining = 0; // track current remaining
  int total = 0; // track total remaining
  int start = 0;

  for (int i = 0; i < gas.length; i++) {
    int remaining = gas[i] - cost[i];

    //if sum remaining of (i-1) >= 0, continue
    if (sumRemaining >= 0) {
      sumRemaining += remaining;
    //otherwise, reset start index to be current
    } else {
      sumRemaining = remaining;
      start = i;
    }
    total += remaining;
  }

  if (total >= 0){
    return start;
```

```
  }else{
    return -1;
  }
}
```

53 Candy

Problem

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?

Java Solution

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.

```
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;
}
```

54 Jump Game

Problem

Given an array of non-negative integers, you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position. Determine if you are able to reach the last index. For example: A = [2,3,1,1,4], return true. A = [3,2,1,0,4], return false.

Java Solution

We can track the maximum length a position can reach. The key to solve this problem is to find 2 conditions: 1) the position can not reach next step (return false) , and 2) the maximum reach the end (return true).

```
public boolean canJump(int[] A) {
   if(A.length <= 1)
      return true;

   int max = A[0];

   for(int i=0; i<A.length; i++){
      //if not enough to go to next
      if(max <= i && A[i] == 0)
         return false;

      //update max
```

```
        if(i + A[i] > max){
            max = i + A[i];
        }

        //max is enough to reach the end
        if(max >= A.length-1)
            return true;
    }

    return false;
}
```

55 Pascal's Triangle

Problem Given numRows, generate the first numRows of Pascal's triangle. For example, given numRows = 5, the result should be:

```
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

Java Solution

```
public ArrayList<ArrayList<Integer>> generate(int numRows) {
  ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
  if (numRows <= 0)
    return result;

  ArrayList<Integer> pre = new ArrayList<Integer>();
  pre.add(1);
  result.add(pre);

  for (int i = 2; i <= numRows; i++) {
    ArrayList<Integer> cur = new ArrayList<Integer>();
```

```
      cur.add(1); //first
      for (int j = 0; j < pre.size() - 1; j++) {
        cur.add(pre.get(j) + pre.get(j + 1)); //middle
      }
      cur.add(1);//last

      result.add(cur);
      pre = cur;
    }

    return result;
}
```

56 Container With Most Water

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.

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.

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;
}
```

57 Count and Say

Problem

The count-and-say sequence is the sequence of integers beginning as follows: 1, 11, 21, 1211, 111221, ...

```
1 is read off as "one 1" or 11.
11 is read off as "two 1s" or 21.
21 is read off as "one 2, then one 1" or 1211.
```

Given an integer n, generate the nth sequence.

Java Solution

The problem can be solved by using a simple iteration. See Java solution below:

```
public String countAndSay(int n) {
  if (n <= 0)
    return null;

  String result = "1";
```

```
    int i = 1;

    while (i < n) {
      StringBuilder sb = new StringBuilder();
      int count = 1;
      for (int j = 1; j < result.length(); j++) {
        if (result.charAt(j) == result.charAt(j - 1)) {
          count++;
        } else {
          sb.append(count);
          sb.append(result.charAt(j - 1));
          count = 1;
        }
      }

      sb.append(count);
      sb.append(result.charAt(result.length()  -  1));
      result = sb.toString();
      i++;
    }

    return result;
}
```

58 Repeated DNA Sequences

Problem

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

For example, given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT", return: ["AAAAACCCCC", "CCCCCAAAAA"].

Java Solution

The key to solve this problem is that each of the 4 nucleotides can be stored in 2 bits. So the 10-letter-long sequence can be converted to 20-bits-long integer. The following is a Java solution. You may use an example to manually execute the program and see how it works.

```
public List<String> findRepeatedDnaSequences(String s) {
  List<String> result = new ArrayList<String>();

  int len = s.length();
  if (len < 10) {
    return result;
  }

  Map<Character, Integer> map = new HashMap<Character, Integer>();
  map.put('A', 0);
  map.put('C', 1);
  map.put('G', 2);
  map.put('T', 3);

  Set<Integer> temp = new HashSet<Integer>();
  Set<Integer> added = new HashSet<Integer>();

  int hash = 0;
  for (int i = 0; i < len; i++) {
    if (i < 9) {
      //each ACGT fit 2 bits, so left shift 2
      hash = (hash << 2) + map.get(s.charAt(i));
    } else {
      hash = (hash << 2) + map.get(s.charAt(i));
      //make length of hash to be 20
      hash = hash & (1 << 20) - 1;

      if (temp.contains(hash) && !added.contains(hash)) {
        result.add(s.substring(i - 9, i + 1));
        added.add(hash); //track added
      } else {
        temp.add(hash);
      }
    }

  }

  return result;
}
```

59 Add Two Numbers

The problem:

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list. Input: (2 ->4 ->3) + (5 ->6 ->4) Output: 7 ->0 ->8

Thoughts

This is a simple problem. It can be solved by doing the following:

- Use a flag to mark if previous sum is >= 10
- Handle the situation that one list is longer than the other
- Correctly move the 3 pointers p1, p2 and p3 which pointer to two input lists and one output list

This leads to solution 1.

Solution 1

```
// Definition for singly-linked list.
public class ListNode {
   int val;
   ListNode next;
   ListNode(int x) {
      val = x;
      next = null;
   }
}

public class Solution {
   public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

      ListNode p1 = l1;
      ListNode p2 = l2;

      ListNode newHead = new ListNode(0);
      ListNode p3 = newHead;

      int val;//store sum

      boolean flag = false;//flag if greater than 10

      while(p1 != null || p2 != null){
         //both p1 and p2 have value
         if(p1 != null && p2 != null){

            if(flag){
               val = p1.val + p2.val + 1;
            }else{
```

```
            val = p1.val + p2.val;
        }

        //if sum >= 10
        if(val >= 10 ){
            flag = true;

        //if sum < 10
        }else{
            flag = false;
        }

        p3.next = new ListNode(val%10);
        p1 = p1.next;
        p2 = p2.next;
    //p1 is null, because p2 is longer
    }else if(p2 != null){

        if(flag){
            val = p2.val + 1;
            if(val >= 10){
                flag = true;
            }else{
                flag = false;
            }
        }else{
            val = p2.val;
            flag = false;
        }

        p3.next = new ListNode(val%10);
        p2 = p2.next;

    ////p2 is null, because p1 is longer
    }else if(p1 != null){

        if(flag){
            val = p1.val + 1;
            if(val >= 10){
                flag = true;
            }else{
                flag = false;
            }
        }else{
            val = p1.val;
            flag = false;
        }

        p3.next = new ListNode(val%10);
        p1 = p1.next;
```

```
            }

            p3 = p3.next;
        }

        //handle situation that same length final sum >=10
        if(p1 == null && p2 == null && flag){
            p3.next = new ListNode(1);
        }

        return newHead.next;
    }
}
```

The hard part is how to make the code more readable. Adding some internal comments and refactoring some code are helpful.

Solution 2

There is nothing wrong with solution 1, but the code is not readable. We can refactor the code and make it much shorter and cleaner.

```
public class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
      int carry =0;

        ListNode newHead = new ListNode(0);
        ListNode p1 = l1, p2 = l2, p3=newHead;

        while(p1 != null || p2 != null){
            if(p1 != null){
                carry += p1.val;
                p1 = p1.next;
            }

            if(p2 != null){
                carry += p2.val;
                p2 = p2.next;
            }

            p3.next = new ListNode(carry%10);
            p3 = p3.next;
            carry /= 10;
        }

        if(carry==1)
            p3.next=new ListNode(1);

        return newHead.next;
```

```
    }
}
```

Exactly the same thing!

Quesion

What is the digits are stored in regular order instead of reversed order?

Answer: We can simple reverse the list, calculate the result, and reverse the result.

60 Reorder List

The problem:

Given a singly linked list L: L0→L1→ ... →Ln-1→Ln, reorder it to: L0→Ln→L1→Ln-1→L2→Ln-2→...

For example, given 1,2,3,4, reorder it to 1,4,2,3. You must do this in-place without altering the nodes' values.

Thoughts

This problem is not straightforward, because it requires "in-place" operations. That means we can only change their pointers, not creating a new list.

Solution

This problem can be solved by doing the following:

- Break list in the middle to two lists (use fast & slow pointers)
- Reverse the order of the second list
- Merge two list back together

The following code is a complete runnable class with testing.

```
//Class definition of ListNode
class ListNode {
  int val;
  ListNode next;

  ListNode(int x) {
    val = x;
    next = null;
  }
```

```
}

public class ReorderList {

  public static void main(String[] args) {
    ListNode n1 = new ListNode(1);
    ListNode n2 = new ListNode(2);
    ListNode n3 = new ListNode(3);
    ListNode n4 = new ListNode(4);
    n1.next = n2;
    n2.next = n3;
    n3.next = n4;

    printList(n1);

    reorderList(n1);

    printList(n1);
  }

  public static void reorderList(ListNode head) {

    if (head != null && head.next != null) {

      ListNode slow = head;
      ListNode fast = head;

      //use a fast and slow pointer to break the link to two parts.
      while (fast != null && fast.next != null && fast.next.next!= null) {
        //why need third/second condition?
        System.out.println("pre "+slow.val + " " + fast.val);
        slow = slow.next;
        fast = fast.next.next;
        System.out.println("after " + slow.val + " " + fast.val);
      }

      ListNode second = slow.next;
      slow.next = null;// need to close first part

      // now should have two lists: head and fast

      // reverse order for second part
      second = reverseOrder(second);

      ListNode p1 = head;
      ListNode p2 = second;

      //merge two lists here
      while (p2 != null) {
        ListNode temp1 = p1.next;
```

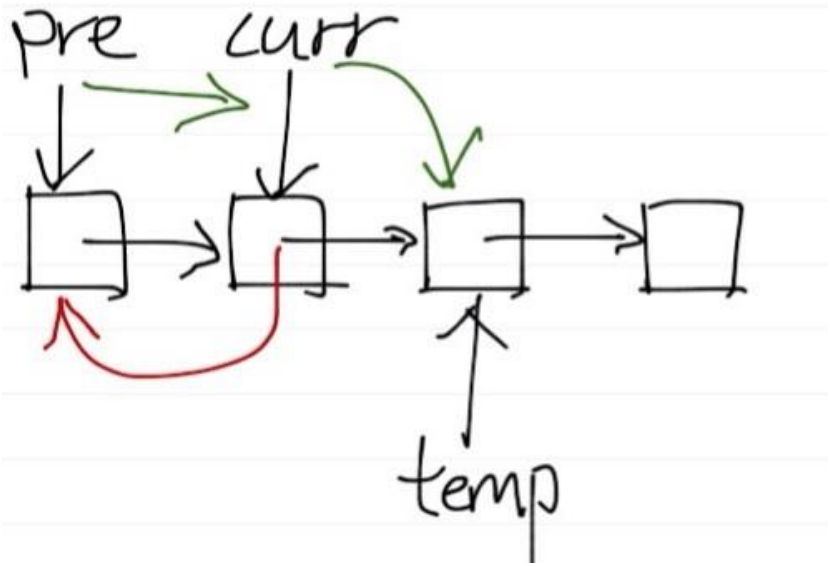
```
        ListNode temp2 = p2.next;

        p1.next = p2;
        p2.next = temp1;

        p1 = temp1;
        p2 = temp2;
      }
    }
  }

  public static ListNode reverseOrder(ListNode head) {

    if (head == null || head.next == null) {
      return head;
    }

    ListNode pre = head;
    ListNode curr = head.next;

    while (curr != null) {
      ListNode temp = curr.next;
      curr.next = pre;
      pre = curr;
      curr = temp;
    }

    // set head node's next
    head.next = null;

    return pre;
  }

  public static void printList(ListNode n) {
    System.out.println("----------");
    while (n != null) {
      System.out.print(n.val);
      n = n.next;
    }
    System.out.println();
  }
}
```

Takeaway Messages from This Problem

The three steps can be used to solve other problems of linked list. A little diagram may help better understand them.

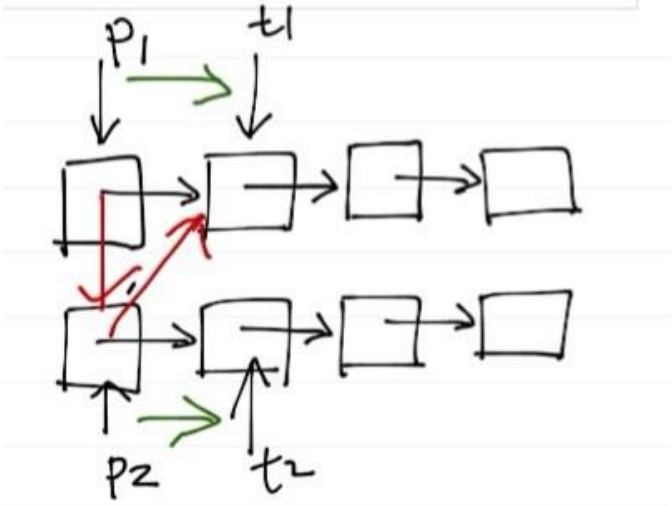
Reverse List:

```
ListNode pre = head;
ListNode curr = head.next;

while (curr != null) {
        ListNode temp = curr.next;
        curr.next = pre;
        pre = curr;
        curr = temp;
}

head.next = null;
```



Merge List:

```
ListNode p1 = head;
ListNode p2 = second;

//merge two lists here
while (p2 != null) {
        ListNode temp1 = p1.next;
        ListNode temp2 = p2.next;

        p1.next = p2;
        p2.next = temp1;

        p1 = temp1;
        p2 = temp2;
}
```



61 Linked List Cycle

Leetcode Problem: Linked List Cycle

Given a linked list, determine if it has a cycle in it.

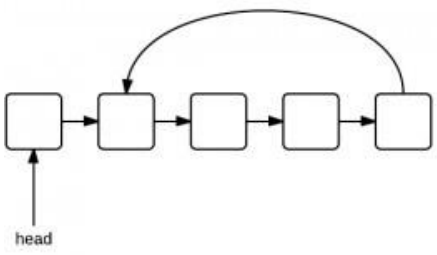
Naive Approach

```
class ListNode {
    int val;
```

```
    ListNode next;
    ListNode(int x) {
       val = x;
       next = null;
    }
 }

public class Solution {
   public boolean hasCycle(ListNode head) {
      ListNode p = head;

      if(head == null)
         return false;

      if(p.next == null)
         return false;

      while(p.next != null){
         if(head == p.next){
            return true;
         }
         p = p.next;
      }

      return false;
   }
}
```

Result:

Submission Result: Time Limit Exceeded Last executed input: 3,2,0,-4, tail connects to node index 1



Accepted Approach

Use fast and low pointer. The advantage about fast/slow pointers is that when a circle is located, the fast one will catch the slow one for sure.

```
public class Solution {
   public boolean hasCycle(ListNode head) {
      ListNode fast = head;
      ListNode slow = head;

      if(head == null)
         return false;

      if(head.next == null)
         return false;

      while(fast != null && fast.next != null){
         slow = slow.next;
         fast = fast.next.next;

         if(slow == fast)
            return true;
      }

      return false;
   }
}
```

62 Copy List with Random Pointer

Problem:

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null. Return a deep copy of the list.

Some Thoughts

We can solve this problem by doing the following steps:

- copy every node, i.e., duplicate every node, and insert it to the list
- copy random pointers for all newly created nodes
- break the list to two

First Attempt (Wrong)

What is wrong with the following code?

```
/**
 * Definition for singly-linked list with a random pointer.
 * class RandomListNode {
 *    int label;
 *    RandomListNode next, random;
 *    RandomListNode(int x) { this.label = x; }
 * };
 */
public class Solution {
   public RandomListNode copyRandomList(RandomListNode head) {

      if(head == null)
         return null;

      RandomListNode p = head;

      //copy every node and insert to list
      while(p != null){
         RandomListNode copy = new RandomListNode(p.label);
         copy.next = p.next;
         p.next = copy;
         p = copy.next;
      }

      //copy random pointer for each new node
      p = head;
      while(p != null){
         p.next.random = p.random.next;//p.random can be null, so need null
             checking here!
         p = p.next.next;
      }

      //break list to two
      p = head;
      while(p !=  null){
         p.next = p.next.next;
         p = p.next;//point to the wrong node now!
      }

      return head.next;
   }
}
```

The code above seems totally fine. It follows the steps designed. But it has run-time errors. Why?

The problem is in the parts of copying random pointer and breaking list.

Correct Solution

```
public RandomListNode copyRandomList(RandomListNode head) {

  if (head == null)
    return null;

  RandomListNode p = head;

  // copy every node and insert to list
  while (p != null) {
    RandomListNode copy = new RandomListNode(p.label);
    copy.next = p.next;
    p.next = copy;
    p = copy.next;
  }

  // copy random pointer for each new node
  p = head;
  while (p != null) {
    if (p.random != null)
      p.next.random = p.random.next;
    p = p.next.next;
  }

  // break list to two
  p = head;
  RandomListNode newHead = head.next;
  while (p != null) {
    RandomListNode temp = p.next;
    p.next = temp.next;
    if (temp.next != null)
      temp.next = temp.next.next;
    p = p.next;
  }

  return newHead;
}
```

The break list part above move pointer 2 steps each time, you can also move one at a time which is simpler, like the following:

```
while(p != null && p.next != null){
   RandomListNode temp = p.next;
   p.next = temp.next;
   p = temp;
}
```

Correct Solution Using HashMap

From Xiaomeng's comment below, we can use a HashMap which makes it simpler.

```
public RandomListNode copyRandomList(RandomListNode head) {
  if (head == null)
    return null;
  HashMap<RandomListNode, RandomListNode> map = new HashMap<RandomListNode,
      RandomListNode>();
  RandomListNode newHead = new RandomListNode(head.label);

  RandomListNode p = head;
  RandomListNode q = newHead;
  map.put(head, newHead);

  p = p.next;
  while (p != null) {
    RandomListNode temp = new RandomListNode(p.label);
    map.put(p, temp);
    q.next = temp;
    q = temp;
    p = p.next;
  }

  p = head;
  q = newHead;
  while (p != null) {
    if (p.random != null)
      q.random = map.get(p.random);
    else
      q.random = null;

    p = p.next;
    q = q.next;
  }

  return newHead;
}
```

63 Merge Two Sorted Lists

Problem:

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

Key to solve this problem

The key to solve the problem is defining a fake head. Then compare the first elements from each list. Add the smaller one to the merged list. Finally, when one of them is empty, simply append it to the merged list, since it is already sorted.

Java Solution

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *    int val;
 *    ListNode next;
 *    ListNode(int x) {
 *       val = x;
 *       next = null;
 *    }
 * }
 */
public class Solution {
   public ListNode mergeTwoLists(ListNode l1, ListNode l2) {

      ListNode p1 = l1;
      ListNode p2 = l2;

      ListNode fakeHead = new ListNode(0);
      ListNode p = fakeHead;

      while(p1 != null && p2 != null){
        if(p1.val <= p2.val){
           p.next = p1;
           p1 = p1.next;
        }else{
           p.next = p2;
           p2 = p2.next;
        }

        p = p.next;
      }

      if(p1 != null)
         p.next = p1;
      if(p2 != null)
         p.next = p2;

      return fakeHead.next;
   }
}
```

64 Merge k Sorted Lists

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Thoughts

The simplest solution is using PriorityQueue. The elements of the priority queue are ordered according to their natural ordering, or by a comparator provided at the construction time (in this case).

Java Solution

```
import java.util.ArrayList;
import java.util.Comparator;
import  java.util.PriorityQueue;

// Definition for singly-linked list.
class ListNode {
  int val;
  ListNode next;

  ListNode(int x) {
    val = x;
    next = null;
  }
}

public class Solution {
  public ListNode mergeKLists(ArrayList<ListNode> lists) {
    if (lists.size() == 0)
      return null;

    //PriorityQueue is a sorted queue
    PriorityQueue<ListNode> q = new PriorityQueue<ListNode>(lists.size(),
        new Comparator<ListNode>() {
          public int compare(ListNode a, ListNode b) {
            if (a.val > b.val)
              return 1;
            else if(a.val == b.val)
              return 0;
            else
```

```
                return -1;
            }
        });

    //add first node of each list to the queue
    for (ListNode list : lists) {
      if (list != null)
        q.add(list);
    }

    ListNode head = new ListNode(0);
    ListNode p = head; // serve as a pointer/cursor

    while (q.size() > 0) {
      ListNode temp = q.poll();
      //poll() retrieves and removes the head of the queue - q.
      p.next = temp;

      //keep adding next element of each list
      if (temp.next != null)
        q.add(temp.next);

      p = p.next;
    }

    return head.next;
  }
}
```

Time: log(k) * n. k is number of list and n is number of total elements.

65 Remove Duplicates from Sorted List

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example,

```
Given 1->1->2, return 1->2.
Given 1->1->2->3->3, return 1->2->3.
```

Thoughts

The key of this problem is using the right loop condition. And change what is necessary in each loop. You can use different iteration conditions like the following 2

solutions.

Solution 1

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *    int val;
 *    ListNode next;
 *    ListNode(int x) {
 *       val = x;
 *       next = null;
 *    }
 * }
 */
public class Solution {
   public ListNode deleteDuplicates(ListNode head) {
      if(head == null || head.next == null)
         return head;

      ListNode prev = head;
      ListNode p = head.next;

      while(p != null){
         if(p.val == prev.val){
            prev.next = p.next;
            p = p.next;
            //no change prev
         }else{
            prev = p;
            p = p.next;
         }
      }

      return head;
   }
}
```

Solution 2

```
public class Solution {
   public ListNode deleteDuplicates(ListNode head) {
      if(head == null || head.next == null)
         return head;

      ListNode p = head;
```

```
        while( p!= null && p.next != null){
            if(p.val == p.next.val){
                p.next = p.next.next;
            }else{
                p = p.next;
            }
        }

        return head;
    }
}
```

66 Partition List

Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

For example, Given 1->4->3->2->5->2 and x = 3, return 1->2->2->4->3->5.

Naive Solution (Wrong)

The following is a solution I write at the beginning. It contains a trivial problem, but it took me a long time to fix it.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode partition(ListNode head, int x) {
        if(head == null) return null;

        ListNode fakeHead1 = new ListNode(0);
        ListNode fakeHead2 = new ListNode(0);
```

```
        fakeHead1.next = head;

        ListNode p = head;
        ListNode prev = fakeHead1;
        ListNode p2 = fakeHead2;

        while(p != null){
            if(p.val < 3){
                p = p.next;
                prev = prev.next;
            }else{
                prev.next = p.next;
                p2.next = p;
                p = prev.next;
                p2 = p2.next;
            }
        }

        p.next = fakeHead2.next;
        return fakeHead1.next;
    }
}
```

Correct Solution

The problem of the first solution is that the last node's next element should be set to null.

```
public class Solution {
    public ListNode partition(ListNode head, int x) {
        if(head == null) return null;

        ListNode fakeHead1 = new ListNode(0);
        ListNode fakeHead2 = new ListNode(0);
        fakeHead1.next = head;

        ListNode p = head;
        ListNode prev = fakeHead1;
        ListNode p2 = fakeHead2;

        while(p != null){
            if(p.val < x){
                p = p.next;
                prev = prev.next;
            }else{

                p2.next = p;
                prev.next = p.next;
```

```
            p = prev.next;
            p2 = p2.next;
        }
    }

    // close the list
    p2.next = null;

    prev.next = fakeHead2.next;

    return fakeHead1.next;
  }
}
```

67 LRU Cache

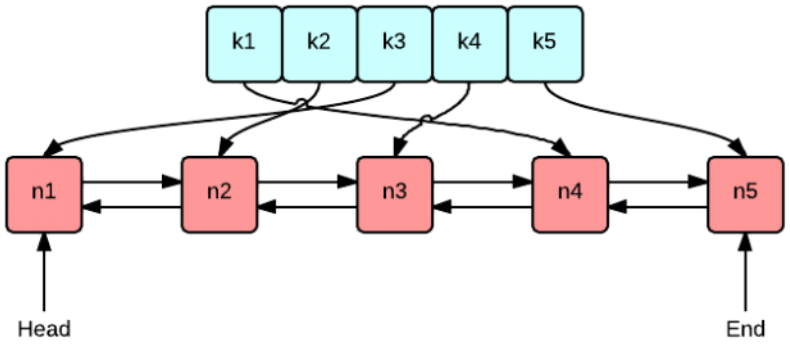
Problem

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1. set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

Java Solution

The key to solve this problem is using a double linked list which enables us to quickly move nodes.




```
import java.util.HashMap;

public class LRUCache {
  private HashMap<Integer, DoubleLinkedListNode> map
    = new HashMap<Integer, DoubleLinkedListNode>();
  private DoubleLinkedListNode head;
  private DoubleLinkedListNode end;
  private int capacity;
  private int len;

  public LRUCache(int capacity) {
    this.capacity = capacity;
    len = 0;
  }

  public int get(int key) {
    if (map.containsKey(key)) {
      DoubleLinkedListNode latest = map.get(key);
      removeNode(latest);
      setHead(latest);
      return latest.val;
    } else {
      return -1;
    }
  }

  public void removeNode(DoubleLinkedListNode node) {
    DoubleLinkedListNode cur = node;
    DoubleLinkedListNode pre = cur.pre;
    DoubleLinkedListNode post = cur.next;

    if (pre != null) {
      pre.next = post;
    } else {
      head = post;
    }
```

```
    if (post != null) {
      post.pre = pre;
    } else {
      end = pre;
    }
  }

  public void setHead(DoubleLinkedListNode node) {
    node.next = head;
    node.pre = null;
    if (head != null) {
      head.pre = node;
    }

    head = node;
    if (end == null) {
      end = node;
    }
  }

  public void set(int key, int value) {
    if (map.containsKey(key)) {
      DoubleLinkedListNode oldNode = map.get(key);
      oldNode.val = value;
      removeNode(oldNode);
      setHead(oldNode);
    } else {
      DoubleLinkedListNode newNode =
        new DoubleLinkedListNode(key, value);
      if (len < capacity) {
        setHead(newNode);
        map.put(key, newNode);
        len++;
      } else {
        map.remove(end.key);
        end = end.pre;
        if (end != null) {
          end.next = null;
        }

        setHead(newNode);
        map.put(key, newNode);
      }
    }
  }
}

class DoubleLinkedListNode {
  public int val;
```

```
  public int key;
  public DoubleLinkedListNode pre;
  public DoubleLinkedListNode next;

  public DoubleLinkedListNode(int key, int value) {
    val = value;
    this.key = key;
  }
}
```

68 Intersection of Two Linked Lists

Problem

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:

```
A:        a1 -> a2
                ->
                  c1 -> c2 -> c3
                ->
B:    b1 -> b2 -> b3
```

begin to intersect at node c1.

Java Solution

First calculate the length of two lists and find the difference. Then start from the longer list at the diff offset, iterate though 2 lists and find the node.

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *    int val;
 *    ListNode next;
 *    ListNode(int x) {
 *       val = x;
 *       next = null;
 *    }
 * }
 */
public class Solution {
```

```
   public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
      int len1 = 0;
      int len2 = 0;
      ListNode p1=headA, p2=headB;
      if (p1 == null || p2 == null)
         return null;

      while(p1 != null){
         len1++;
         p1 = p1.next;
      }
      while(p2 !=null){
         len2++;
         p2 = p2.next;
      }

      int diff = 0;
      p1=headA;
      p2=headB;

      if(len1 > len2){
         diff = len1-len2;
         int i=0;
         while(i<diff){
            p1 = p1.next;
            i++;
         }
      }else{
         diff = len2-len1;
         int i=0;
         while(i<diff){
            p2 = p2.next;
            i++;
         }
      }

      while(p1 != null && p2 != null){
         if(p1.val == p2.val){
            return p1;
         }else{

         }
         p1 = p1.next;
         p2 = p2.next;
      }

      return null;
   }
}
```

69 Java PriorityQueue Class Example

In Java, the PriorityQueue class is implemented as a priority heap. Heap is an important data structure in computer science. For a quick overview of heap, here is a very good tutorial.

Simple Example

The following examples shows the basic operations of PriorityQueue such as offer(), peek(), poll(), and size().

```
import java.util.Comparator;
import  java.util.PriorityQueue;

public class PriorityQueueTest {

  static class PQsort implements Comparator<Integer> {

    public int compare(Integer one, Integer two) {
      return two - one;
    }
  }

  public static void main(String[] args) {
    int[] ia = { 1, 10, 5, 3, 4, 7, 6, 9, 8 };
    PriorityQueue<Integer> pq1 = new PriorityQueue<Integer>();

    // use offer() method to add elements to the PriorityQueue pq1
    for (int x : ia) {
      pq1.offer(x);
    }

    System.out.println("pq1: " + pq1);

    PQsort pqs = new PQsort();
    PriorityQueue<Integer> pq2 = new PriorityQueue<Integer>(10, pqs);
    // In this particular case, we can simply use Collections.reverseOrder()
    // instead of self-defined comparator
    for (int x : ia) {
      pq2.offer(x);
    }

    System.out.println("pq2: " + pq2);
```

```
        // print size
        System.out.println("size: " + pq2.size());
        // return highest priority element in the queue without removing it
        System.out.println("peek: " + pq2.peek());
        // print size
        System.out.println("size: " + pq2.size());
        // return highest priority element and removes it from the queue
        System.out.println("poll: " + pq2.poll());
        // print size
        System.out.println("size: " + pq2.size());

        System.out.print("pq2: " + pq2);

    }
}
```

Output:

```
pq1: [1, 3, 5, 8, 4, 7, 6, 10, 9]
pq2: [10, 9, 7, 8, 3, 5, 6, 1, 4]
size: 9
peek: 10
size: 9
poll: 10
size: 8
pq2: [9, 8, 7, 4, 3, 5, 6, 1]
```

Example of Solving Problems Using PriorityQueue

Merging k sorted list.

For more details about PriorityQueue, please go to doc.

70 Solution for Binary Tree Preorder Traversal in Java

Preorder binary tree traversal is a classic interview problem about trees. The key to solve this problem is to understand the following:

- What is preorder? (parent node is processed before its children)
- Use Stack from Java Core library

It is not obvious what preorder for some strange cases. However, if you draw a stack and manually execute the program, how each element is pushed and popped is

obvious.

The key to solve this problem is using a stack to store left and right children, and push right child first so that it is processed after the left child.

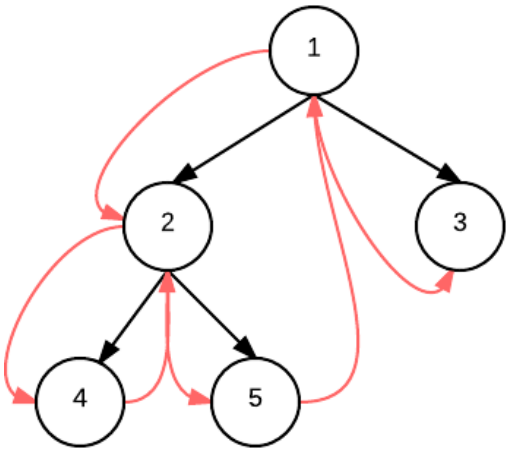
```
public class TreeNode {
   int val;
   TreeNode left;
   TreeNode right;
   TreeNode(int x) { val = x; }
}

public class Solution {
   public ArrayList<Integer> preorderTraversal(TreeNode root) {
      ArrayList<Integer> returnList = new ArrayList<Integer>();

      if(root == null)
         return returnList;

      Stack<TreeNode> stack = new Stack<TreeNode>();
      stack.push(root);

      while(!stack.empty()){
         TreeNode n = stack.pop();
         returnList.add(n.val);

         if(n.right != null){
            stack.push(n.right);
         }
         if(n.left != null){
            stack.push(n.left);
         }

      }
      return returnList;
   }
}
```

71 Solution of Binary Tree Inorder Traversal in Java

The key to solve inorder traversal of binary tree includes the following:

- The order of "inorder" is: left child ->parent ->right child

- Use a stack to track nodes
- Understand when to push node into the stack and when to pop node out of the stack




```
//Definition for binary tree
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
 }

public class Solution {
   public ArrayList<Integer> inorderTraversal(TreeNode root) {
      // IMPORTANT: Please reset any member data you declared, as
      // the same Solution instance will be reused for each test case.
       ArrayList<Integer> lst = new ArrayList<Integer>();

      if(root == null)
         return lst;

      Stack<TreeNode> stack = new Stack<TreeNode>();
      //define a pointer to track nodes
      TreeNode p = root;

      while(!stack.empty() || p != null){

         // if it is not null, push to stack
         //and go down the tree to left
         if(p != null){
            stack.push(p);
            p = p.left;
```

```
            // if no left child
            // pop stack, process the node
            // then let p point to the right
            }else{
               TreeNode t = stack.pop();
               lst.add(t.val);
               p = t.right;
            }
         }

         return lst;
      }
}
```

72 Solution of Iterative Binary Tree Postorder Traversal in Java

The key to to iterative postorder traversal is the following:

- The order of "Postorder" is: left child ->right child ->parent node.
- Find the relation between the previously visited node and the current node
- Use a stack to track nodes

As we go down the tree, check the previously visited node. If it is the parent of the current node, we should add current node to stack. When there is no children for current node, pop it from stack. Then the previous node become to be under the current node for next loop.

```
//Definition for binary tree
public class TreeNode {
   int val;
   TreeNode left;
   TreeNode right;
   TreeNode(int x) { val = x; }
}


public class Solution {
   public ArrayList<Integer> postorderTraversal(TreeNode root) {

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

      if(root == null)
```

```
            return lst;

        Stack<TreeNode> stack = new Stack<TreeNode>();
        stack.push(root);

        TreeNode prev = null;
        while(!stack.empty()){
            TreeNode curr = stack.peek();

            // go down the tree.
            //check if current node is leaf, if so, process it and pop stack,
            //otherwise, keep going down
            if(prev == null || prev.left == curr || prev.right == curr){
                //prev == null is the situation for the root node
                if(curr.left != null){
                    stack.push(curr.left);
                }else if(curr.right != null){
                    stack.push(curr.right);
                }else{
                    stack.pop();
                    lst.add(curr.val);
                }

            //go up the tree from left node
            //need to check if there is a right child
            //if yes, push it to stack
            //otherwise, process parent and pop stack
            }else if(curr.left == prev){
                if(curr.right != null){
                    stack.push(curr.right);
                }else{
                    stack.pop();
                    lst.add(curr.val);
                }

            //go up the tree from right node
            //after coming back from right node, process parent node and pop
                stack.
            }else if(curr.right == prev){
                stack.pop();
                lst.add(curr.val);
            }

            prev = curr;
        }

        return lst;
    }
}
```

73 Validate Binary Search Tree

Problem:

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Thoughts about This Problem

All values on the left sub tree must be less than root, and all values on the right sub tree must be greater than root.

Java Solution

```
// Definition for binary tree
class TreeNode {
  int val;
  TreeNode left;
  TreeNode right;

  TreeNode(int x) {
    val = x;
  }
}

public class Solution {

  public static boolean isValidBST(TreeNode root) {
    return validate(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
  }

  public static boolean validate(TreeNode root, int min, int max) {
    if (root == null) {
      return true;
    }

    // not in range
```

```
    if (root.val <= min || root.val >= max) {
      return false;
    }

    // left subtree must be < root.val && right subtree must be > root.val
    return validate(root.left, min, root.val) && validate(root.right,
        root.val, max);
  }
}
```

74 Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

For example, Given

```
1
       / \
      2   5
     / \   \
    3   4   6
```

The flattened tree should look like:

```
1
    \
     2
      \
       3
        \
         4
          \
           5
            \
             6
```

Thoughts

Go down through the left, when right is not null, push right to stack.

Java Solution

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *    int val;
 *    TreeNode left;
 *    TreeNode right;
 *    TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
   public void flatten(TreeNode root) {
      Stack<TreeNode> stack = new Stack<TreeNode>();
      TreeNode p = root;

      while(p != null || !stack.empty()){

         if(p.right != null){
            stack.push(p.right);
         }

         if(p.left != null){
            p.right = p.left;
            p.left = null;
         }else if(!stack.empty()){
            TreeNode temp = stack.pop();
            p.right=temp;
         }

         p = p.right;
      }
   }
}
```

75 Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example: Given the below binary tree and sum = 22,

```
5
          / \
         4   8
        / / \
       11 13 4
```

```
      / \     \
     7   2     1
```

return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

Java Solution 1 - Using Queue

Add all node to a queue and store sum value of each node to another queue. When it is a leaf node, check the stored sum value.

For the tree above, the queue would be: 5 - 4 - 8 - 11 - 13 - 4 - 7 - 2 - 1. It will check node 13, 7, 2 and 1. This is a typical breadth first search(BFS) problem.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *    int val;
 *    TreeNode left;
 *    TreeNode right;
 *    TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
   public boolean hasPathSum(TreeNode root, int sum) {
      if(root == null) return false;

      LinkedList<TreeNode> nodes = new LinkedList<TreeNode>();
      LinkedList<Integer> values = new LinkedList<Integer>();

      nodes.add(root);
      values.add(root.val);

      while(!nodes.isEmpty()){
         TreeNode curr = nodes.poll();
         int sumValue = values.poll();

         if(curr.left == null && curr.right == null && sumValue==sum){
            return true;
         }

         if(curr.left != null){
            nodes.add(curr.left);
            values.add(sumValue+curr.left.val);
         }

         if(curr.right != null){
            nodes.add(curr.right);
            values.add(sumValue+curr.right.val);
         }
      }
```

```
      return false;
    }
}
```

Java Solution 2 - Recursion

```
public boolean hasPathSum(TreeNode root, int sum) {
  if (root == null)
    return false;
  if (root.val == sum && (root.left == null && root.right == null))
    return true;

  return hasPathSum(root.left, sum - root.val)
      || hasPathSum(root.right, sum - root.val);
}
```

Thanks to nebulaliang, this solution is wonderful!

76 Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

Throughts

This problem can be illustrated by using a simple example.

```
in-order: 4 2 5 (1) 6 7 3 8
post-order: 4 5 2 6 7 8 3 (1)
```

From the post-order array, we know that last element is the root. We can find the root in in-order array. Then we can identify the left and right sub-trees of the root from in-order array.

Using the length of left sub-tree, we can identify left and right sub-trees in post-order array. Recursively, we can build up the tree.

Java Solution

```
//Definition for binary tree
```

```
 public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
 }

public class Solution {
   public TreeNode buildTree(int[] inorder, int[] postorder) {
      int inStart = 0;
      int inEnd = inorder.length-1;
      int postStart =0;
      int postEnd = postorder.length-1;

      return buildTree(inorder, inStart, inEnd, postorder, postStart,
          postEnd);
   }

   public TreeNode buildTree(int[] inorder, int inStart, int inEnd,
                     int[] postorder, int postStart, int postEnd){
      if(inStart > inEnd || postStart > postEnd)
         return null;

      int rootValue = postorder[postEnd];
      TreeNode root = new TreeNode(rootValue);

      int k=0;
      for(int i=0; i< inorder.length; i++){
         if(inorder[i]==rootValue){
            k = i;
            break;
         }
      }

      root.left = buildTree(inorder, inStart, k-1, postorder, postStart,
          postStart+k-(inStart+1));
      // Becuase k is not the length, it it need to -(inStart+1) to get the
          length
      root.right = buildTree(inorder, k+1, inEnd, postorder,
          postStart+k-inStart, postEnd-1);
      // postStart+k-inStart = postStart+k-(inStart+1) +1

      return root;
   }
}
```

77 Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

Thoughts

Straightforward! Recursively do the job.

Java Solution

```
// Definition for binary tree
class TreeNode {
  int val;
  TreeNode left;
  TreeNode right;

  TreeNode(int x) {
    val = x;
  }
}

public class Solution {
  public TreeNode sortedArrayToBST(int[] num) {
    if (num.length == 0)
      return null;

    return sortedArrayToBST(num, 0, num.length - 1);
  }

  public TreeNode sortedArrayToBST(int[] num, int start, int end) {
    if (start > end)
      return null;

    int mid = (start + end) / 2;
    TreeNode root = new TreeNode(num[mid]);
    root.left = sortedArrayToBST(num, start, mid - 1);
    root.right = sortedArrayToBST(num, mid + 1, end);

    return root;
  }
}
```

78 Convert Sorted List to Binary Search Tree

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

Thoughts

If you are given an array, the problem is quite straightforward. But things get a little more complicated when you have a singly linked list instead of an array. Now you no longer have random access to an element in O(1) time. Therefore, you need to create nodes bottom-up, and assign them to its parents. The bottom-up approach enables us to access the list in its order at the same time as creating nodes.

Java Solution

```
// Definition for singly-linked list.
class ListNode {
  int val;
  ListNode next;

  ListNode(int x) {
    val = x;
    next = null;
  }
}

// Definition for binary tree
class TreeNode {
  int val;
  TreeNode left;
  TreeNode right;

  TreeNode(int x) {
    val = x;
  }
}

public class Solution {
  static ListNode h;

  public TreeNode sortedListToBST(ListNode head) {
    if (head == null)
      return null;
```

```
    h = head;
    int len = getLength(head);
    return sortedListToBST(0, len - 1);
  }

  // get list length
  public int getLength(ListNode head) {
    int len = 0;
    ListNode p = head;

    while (p != null) {
      len++;
      p = p.next;
    }
    return len;
  }

  // build tree bottom-up
  public TreeNode sortedListToBST(int start, int end) {
    if (start > end)
      return null;

    // mid
    int mid = (start + end) / 2;

    TreeNode left = sortedListToBST(start, mid - 1);
    TreeNode root = new TreeNode(h.val);
    h = h.next;
    TreeNode right = sortedListToBST(mid + 1, end);

    root.left = left;
    root.right = right;

    return root;
  }
}
```

79 Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Thoughts

Need to know LinkedList is a queue. add() and remove() are the two methods to manipulate the queue.

Java Solution

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *    int val;
 *    TreeNode left;
 *    TreeNode right;
 *    TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
   public int minDepth(TreeNode root) {
      if(root == null){
         return 0;
      }

      LinkedList<TreeNode> nodes = new LinkedList<TreeNode>();
      LinkedList<Integer> counts = new LinkedList<Integer>();

      nodes.add(root);
      counts.add(1);

      while(!nodes.isEmpty()){
         TreeNode curr = nodes.remove();
         int count = counts.remove();

         if(curr.left != null){
            nodes.add(curr.left);
            counts.add(count+1);
         }

         if(curr.right != null){
            nodes.add(curr.right);
            counts.add(count+1);
         }

         if(curr.left == null && curr.right == null){
            return count;
         }
      }

      return 0;
   }
```

```
}
```

80 Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree.

For example: Given the below binary tree,

```
1
     / \
    2  3
```

Return 6.

Thoughts

1) Recursively solve this problem 2) Get largest left sum and right sum 2) Compare to the stored maximum

Java Solution 1

```
// Definition for binary tree
class TreeNode {
  int val;
  TreeNode left;
  TreeNode right;

  TreeNode(int x) {
    val = x;
  }
}

public class Solution {
  //store max value
  int max;

  public int maxPathSum(TreeNode root) {
    max = (root == null) ? 0 : root.val;
    findMax(root);
    return max;
  }
```

```
  public int findMax(TreeNode node) {
    if (node == null)
      return 0;

    // recursively get sum of left and right path
    int left = Math.max(findMax(node.left), 0);
    int right = Math.max(findMax(node.right), 0);

    //update maximum here
    max = Math.max(node.val + left + right, max);

    // return sum of largest path of current node
    return node.val + Math.max(left, right);
  }
}
```

Java Solution 2

We can also use an array to store value for recursive methods.

```
public class Solution {
  public int maxPathSum(TreeNode root) {
    int max[] = new int[1];
    max[0] = Integer.MIN_VALUE;
    calculateSum(root, max);
    return max[0];
  }

  public int calculateSum(TreeNode root, int[] max) {
    if (root == null)
      return 0;

    int left = calculateSum(root.left, max);
    int right = calculateSum(root.right, max);

    int current = Math.max(root.val, Math.max(root.val + left, root.val +
        right));

    max[0] = Math.max(max[0], Math.max(current, left + root.val + right));

    return current;
  }
}
```

81 Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

Thoughts

A typical recursive problem for solving tree problems.

Java Solution

```
// Definition for binary tree
class TreeNode {
  int val;
  TreeNode left;
  TreeNode right;

  TreeNode(int x) {
    val = x;
  }
}

public class Solution {
  public boolean isBalanced(TreeNode root) {
    if (root == null)
      return true;

    if (getHeight(root) == -1)
      return false;

    return true;
  }

  public int getHeight(TreeNode root) {
    if (root == null)
      return 0;

    int left = getHeight(root.left);
    int right = getHeight(root.right);

    if (left == -1 || right == -1)
      return -1;

    if (Math.abs(left - right) > 1) {
      return -1;
```

```
    }

    return Math.max(left, right) + 1;

  }
}
```

82 Symmetric Tree

Problem

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree is symmetric:

```
1
  / \
 2   2
/ \ / \
3 4 4 3
```

But the following is not:

```
1
  / \
 2   2
  \   \
  3    3
```

Java Solution - Recursion

This problem can be solve by using a simple recursion. The key is finding the conditions that return false, such as value is not equal, only one node(left or right) has value.

```
public boolean isSymmetric(TreeNode root) {
  if (root == null)
    return true;
  return isSymmetric(root.left, root.right);
}

public boolean isSymmetric(TreeNode l, TreeNode r) {
```

```
  if (l == null && r == null) {
    return true;
  } else if (r == null || l == null) {
    return false;
  }

  if (l.val != r.val)
    return false;

  if (!isSymmetric(l.left, r.right))
    return false;
  if (!isSymmetric(l.right, r.left))
    return false;

  return true;
}
```

83 Clone Graph Java

LeetCode Problem:

> *Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.*

OJ's undirected graph serialization:

Nodes are labeled uniquely.

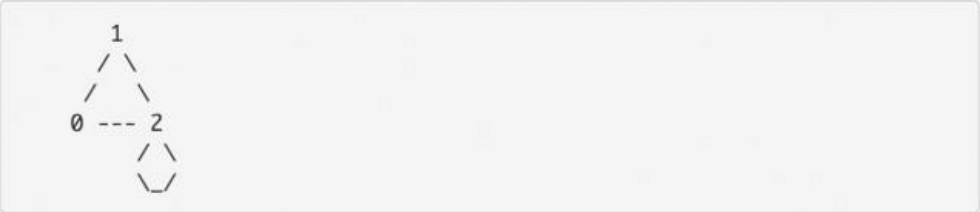
We use `#` as a separator for each node, and `,` as a separator for node label and each neighbor of the node.

As an example, consider the serialized graph `{0,1,2#1,2#2,2}`.

The graph has a total of three nodes, and therefore contains three parts as separated by `#`.

1. First node is labeled as `0`. Connect node `0` to both nodes `1` and `2`.
2. Second node is labeled as `1`. Connect node `1` to node `2`.
3. Third node is labeled as `2`. Connect node `2` to node `2` (itself), thus forming a self-cycle.

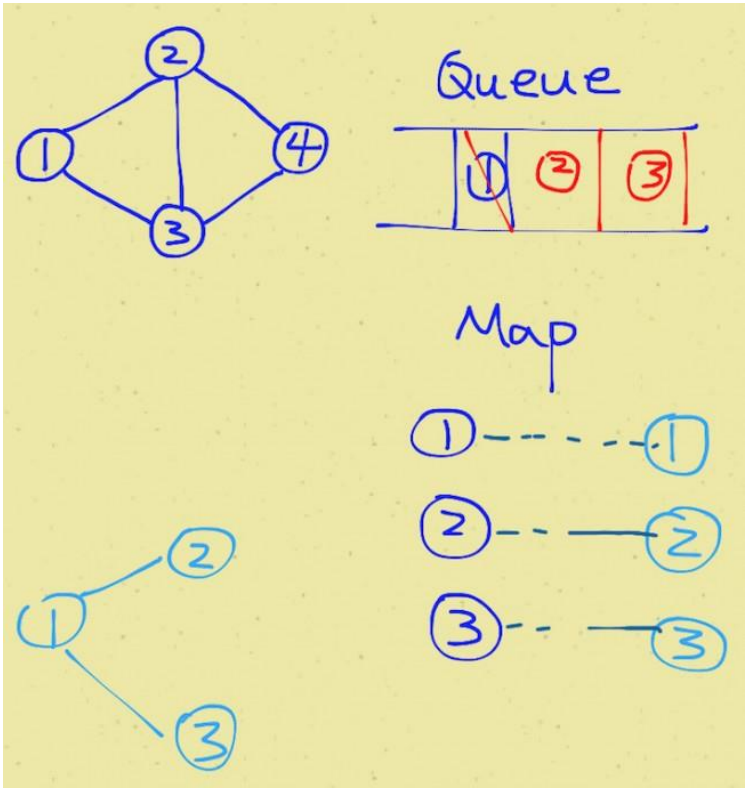
Visually, the graph looks like the following:



Key to Solve This Problem

- A queue is used to do breath first traversal.
- a map is used to store the visited nodes. It is the map between original node and copied node.

It would be helpful if you draw a diagram and visualize the problem.




```
/**
 * Definition for undirected graph.
 * class UndirectedGraphNode {
 *    int label;
 *    ArrayList<UndirectedGraphNode>  neighbors;
 *    UndirectedGraphNode(int x) { label = x; neighbors = new
     ArrayList<UndirectedGraphNode>(); }
 * };
 */
public class Solution {
   public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
      if(node == null)
         return null;

      LinkedList<UndirectedGraphNode> queue = new
          LinkedList<UndirectedGraphNode>();
      HashMap<UndirectedGraphNode, UndirectedGraphNode> map =
                           new
                               HashMap<UndirectedGraphNode,UndirectedGraphNode>();

      UndirectedGraphNode newHead = new UndirectedGraphNode(node.label);

      queue.add(node);
      map.put(node, newHead);
```

```
        while(!queue.isEmpty()){
            UndirectedGraphNode curr = queue.pop();
            ArrayList<UndirectedGraphNode>  currNeighbors  =  curr.neighbors;

            for(UndirectedGraphNode  aNeighbor:  currNeighbors){
                if(!map.containsKey(aNeighbor)){
                    UndirectedGraphNode copy = new
                        UndirectedGraphNode(aNeighbor.label);
                    map.put(aNeighbor,copy);
                    map.get(curr).neighbors.add(copy);
                    queue.add(aNeighbor);
                }else{
                    map.get(curr).neighbors.add(map.get(aNeighbor));
                }
            }

        }
        return newHead;
    }
}
```

84 How Developers Sort in Java?

While analyzing source code of a large number of open source Java projects, I found Java developers frequently sort in two ways. One is using the sort() method of Collections or Arrays, and the other is using sorted data structures, such as TreeMap and TreeSet.

Using sort() Method

If it is a collection, use Collections.sort() method.

```
// Collections.sort
List<ObjectName> list = new ArrayList<ObjectName>();
Collections.sort(list, new Comparator<ObjectName>() {
  public int compare(ObjectName o1, ObjectName o2) {
    return  o1.toString().compareTo(o2.toString());
  }
});
```

If it is an array, use Arrays.sort() method.

```
// Arrays.sort
ObjectName[] arr = new ObjectName[10];
Arrays.sort(arr, new Comparator<ObjectName>() {
  public int compare(ObjectName o1, ObjectName o2) {
    return o1.toString().compareTo(o2.toString());
  }
});
```

This is very convenient if a collection or an array is already set up.

Using Sorted Data Structures

If it is a list or set, use TreeSet to sort.

```
// TreeSet
Set<ObjectName> sortedSet = new TreeSet<ObjectName>(new
    Comparator<ObjectName>() {
  public int compare(ObjectName o1, ObjectName o2) {
    return o1.toString().compareTo(o2.toString());
  }
});
sortedSet.addAll(unsortedSet);
```

If it is a map, use TreeMap to sort. TreeMap is sorted by key.

```
// TreeMap - using String.CASE_INSENSITIVE_ORDER which is a Comparator that
    orders Strings by compareToIgnoreCase
Map<String, Integer> sortedMap = new TreeMap<String,
    Integer>(String.CASE_INSENSITIVE_ORDER);
sortedMap.putAll(unsortedMap);
```

```
//TreeMap - In general, defined comparator
Map<ObjectName, String> sortedMap = new TreeMap<ObjectName, String>(new
    Comparator<ObjectName>() {
  public int compare(ObjectName o1, ObjectName o2) {
    return o1.toString().compareTo(o2.toString());
  }
});
sortedMap.putAll(unsortedMap);
```

This approach is very useful, if you would do a lot of search operations for the collection. The sorted data structure will give time complexity of O(logn), which is lower than O(n).

Bad Practices

There are still bad practices, such as using self-defined sorting algorithm. Take the code below for example, not only the algorithm is not efficient, but also it is not

readable. This happens a lot in different forms of variations.

```
double t;
for (int i = 0; i < 2; i++)
  for (int j = i + 1; j < 3; j++)
    if (r[j] < r[i]) {
      t = r[i];
      r[i] = r[j];
      r[j] = t;
    }
```

85 Solution Merge Sort LinkedList in Java

LeetCode - Sort List:

Sort a linked list in O(n log n) time using constant space complexity.

Keys for solving the problem

- Break the list to two in the middle
- Recursively sort the two sub lists
- Merge the two sub lists

This is my accepted answer for the problem.

```
package algorithm.sort;

class ListNode {
  int val;
  ListNode next;

  ListNode(int x) {
    val = x;
    next = null;
  }
}

public class SortLinkedList {

  // merge sort
  public static ListNode mergeSortList(ListNode head) {

    if (head == null || head.next == null)
      return head;
```

```
    // count total number of elements
    int count = 0;
    ListNode p = head;
    while (p != null) {
      count++;
      p = p.next;
    }

    // break up to two list
    int middle = count / 2;

    ListNode l = head, r = null;
    ListNode p2 = head;
    int countHalf = 0;
    while (p2 != null) {
      countHalf++;
      ListNode next = p2.next;

      if (countHalf == middle) {
        p2.next = null;
        r = next;
      }
      p2 = next;
    }

    // now we have two parts l and r, recursively sort them
    ListNode h1 = mergeSortList(l);
    ListNode h2 = mergeSortList(r);

    // merge together
    ListNode merged = merge(h1, h2);

    return merged;
  }

  public static ListNode merge(ListNode l, ListNode r) {
    ListNode p1 = l;
    ListNode p2 = r;

    ListNode fakeHead = new ListNode(100);
    ListNode pNew = fakeHead;

    while (p1 != null || p2 != null) {

      if (p1 == null) {
        pNew.next = new ListNode(p2.val);
        p2 = p2.next;
        pNew = pNew.next;
      } else if (p2 == null) {
```

```
        pNew.next = new ListNode(p1.val);
        p1 = p1.next;
        pNew = pNew.next;
      } else {
        if (p1.val < p2.val) {
          // if(fakeHead)
          pNew.next = new ListNode(p1.val);
          p1 = p1.next;
          pNew = pNew.next;
        } else if (p1.val == p2.val) {
          pNew.next = new ListNode(p1.val);
          pNew.next.next = new ListNode(p1.val);
          pNew = pNew.next.next;
          p1 = p1.next;
          p2 = p2.next;

        } else {
          pNew.next = new ListNode(p2.val);
          p2 = p2.next;
          pNew = pNew.next;
        }
      }
    }

    // printList(fakeHead.next);
    return fakeHead.next;
  }

  public static void main(String[] args) {
    ListNode n1 = new ListNode(2);
    ListNode n2 = new ListNode(3);
    ListNode n3 = new ListNode(4);

    ListNode n4 = new ListNode(3);
    ListNode n5 = new ListNode(4);
    ListNode n6 = new ListNode(5);

    n1.next = n2;
    n2.next = n3;
    n3.next = n4;
    n4.next = n5;
    n5.next = n6;

    n1 = mergeSortList(n1);

    printList(n1);
  }

  public static void printList(ListNode x) {
    if(x != null){
```

```
        System.out.print(x.val + " ");
        while (x.next != null) {
          System.out.print(x.next.val + " ");
          x = x.next;
        }
        System.out.println();
      }

    }
}
```

Output:
2 3 3 4 4 5

86 Quicksort Array in Java

Quicksort is a divide and conquer algorithm. It first divides a large list into two smaller sub-lists and then recursively sort the two sub-lists. If we want to sort an array without any extra space, Quicksort is a good option. On average, time complexity is O(n log(n)).

The basic step of sorting an array are as follows:

- Select a pivot, normally the middle one
- From both ends, swap elements and make all elements on the left less than the pivot and all elements on the right greater than the pivot
- Recursively sort left part and right part

```
package algorithm.sort;

public class QuickSort {

  public static void main(String[] args) {
    int[] x = { 9, 2, 4, 7, 3, 7, 10 };
    printArray(x);

    int low = 0;
    int high = x.length - 1;

    quickSort(x, low, high);
    printArray(x);
  }

  public static void quickSort(int[] arr, int low, int high) {
```

```
    if (arr == null || arr.length == 0)
      return;

    if (low >= high)
      return;

    //pick the pivot
    int middle = low + (high - low) / 2;
    int pivot = arr[middle];

    //make left < pivot and right > pivot
    int i = low, j = high;
    while (i <= j) {
      while (arr[i] < pivot) {
        i++;
      }

      while (arr[j] > pivot) {
        j--;
      }

      if (i <= j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
        i++;
        j--;
      }
    }

    //recursively sort two sub parts
    if (low < j)
      quickSort(arr, low, j);

    if (high > i)
      quickSort(arr, i, high);
  }

  public static void printArray(int[] x) {
    for (int a : x)
      System.out.print(a + " ");
    System.out.println();
  }
}
```

Output:

9 2 4 7 3 7 10 2 3 4 7 7 9 10

The mistake I made is selecting the middle element. The middle element is not (low+high)/2, but low + (high-low)/2. For other parts of the programs, just follow the

algorithm.

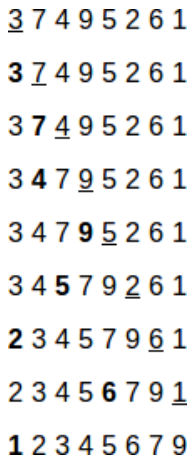
87 Solution Sort a linked list using insertion sort in Java

Insertion Sort List:

Sort a linked list using insertion sort.

This is my accepted answer for LeetCode problem - Sort a linked list using insertion sort in Java. It is a complete program.

Before coding for that, here is an example of insertion sort from wiki. You can get an idea of what is insertion sort.



Code:

```
package algorithm.sort;

class ListNode {
  int val;
  ListNode next;

  ListNode(int x) {
    val = x;
    next = null;
  }
}

public class SortLinkedList {
  public static ListNode insertionSortList(ListNode head) {
```

```
    if (head == null || head.next == null)
      return head;

    ListNode newHead = new ListNode(head.val);
    ListNode pointer = head.next;

    // loop through each element in the list
    while (pointer != null) {
      // insert this element to the new list

      ListNode innerPointer = newHead;
      ListNode next = pointer.next;

      if (pointer.val <= newHead.val) {
        ListNode oldHead = newHead;
        newHead = pointer;
        newHead.next = oldHead;
      } else {
        while (innerPointer.next != null) {

          if (pointer.val > innerPointer.val && pointer.val <=
              innerPointer.next.val) {
            ListNode oldNext = innerPointer.next;
            innerPointer.next = pointer;
            pointer.next = oldNext;
          }

          innerPointer = innerPointer.next;
        }

        if (innerPointer.next == null && pointer.val > innerPointer.val) {
          innerPointer.next = pointer;
          pointer.next = null;
        }
      }

      // finally
      pointer = next;
    }

    return newHead;
  }

  public static void main(String[] args) {
    ListNode n1 = new ListNode(2);
    ListNode n2 = new ListNode(3);
    ListNode n3 = new ListNode(4);

    ListNode n4 = new ListNode(3);
    ListNode n5 = new ListNode(4);
```

```
    ListNode n6 = new ListNode(5);

    n1.next = n2;
    n2.next = n3;
    n3.next = n4;
    n4.next = n5;
    n5.next = n6;

    n1 = insertionSortList(n1);

    printList(n1);

  }

  public static void printList(ListNode x) {
    if(x != null){
      System.out.print(x.val + " ");
      while (x.next != null) {
        System.out.print(x.next.val + " ");
        x = x.next;
      }
      System.out.println();
    }

  }
}
```

Output:
2 3 3 4 4 5

88 Maximum Gap

Problem

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Try to solve it in linear time/space. Return 0 if the array contains less than 2 elements. You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

Java Solution 1 - Sort

A straightforward solution would be sorting the array first (O(nlogn) and then finding the maximum gap. The basic idea is to project each element of the array to an array of

buckets. Each bucket tracks the maximum and minimum elements. Finally, scanning the bucket list, we can get the maximum gap.

The key part is to get the interval:

```
From: interval * (num[i] - min) = 0 and interval * (max -num[i]) = n
interval = num.length / (max - min)
```

See the internal comment for more details.

Java Solution 2 - Bucket Sort

We can use a bucket-sort like algorithm to solve this problem in time of O(n) and space O(n).

```
class Bucket{
    int low;
    int high;
    public Bucket(){
        low = -1;
        high = -1;
    }
}

public int maximumGap(int[] num) {
    if(num == null || num.length < 2){
        return 0;
    }

    int max = num[0];
    int min = num[0];
    for(int i=1; i<num.length; i++){
        max = Math.max(max, num[i]);
        min = Math.min(min, num[i]);
    }

    // initialize an array of buckets
    Bucket[] buckets = new Bucket[num.length+1]; //project to (0 - n)
    for(int i=0; i<buckets.length; i++){
        buckets[i] = new Bucket();
    }

    double interval = (double) num.length / (max - min);
    //distribute every number to a bucket array
    for(int i=0; i<num.length; i++){
        int index = (int) ((num[i] - min) * interval);

        if(buckets[index].low == -1){
            buckets[index].low = num[i];
            buckets[index].high = num[i];
```

```
        }else{
            buckets[index].low = Math.min(buckets[index].low, num[i]);
            buckets[index].high  =  Math.max(buckets[index].high,  num[i]);
        }
    }

    //scan buckets to find maximum gap
    int result = 0;
    int prev = buckets[0].high;
    for(int i=1; i<buckets.length; i++){
        if(buckets[i].low != -1){
            result = Math.max(result, buckets[i].low-prev);
            prev = buckets[i].high;
        }

    }

    return result;
}
```

89 Iteration vs. Recursion in Java

Recursion

Consider the factorial function: n!=n*(n-1)*(n-2)*...*1

There are many ways to compute factorials. One way is that n! is equal to n*(n-1)!. Therefore the program can be directly written as:

Program 1:

```
int factorial (int n) {
    if (n == 1) {
        return 1;
    } else {
        return n*factorial(n-1);
    }
}
```

In order to run this program, the computer needs to build up a chain of multiplications: factorial(n)→ factorial(n-1)→ factorial(n-2)→ ... →factorial(1). Therefore, the computer has to keep track of the multiplications to be performed later on. This type of program, characterized by a chain of operations, is called recursion. Recursion can be further categorized into linear and tree recursion. When the amount of information needed to keep track of the chain of operations grows linearly with the input,

the recursion is called linear recursion. The computation of n! is such a case, because the time required grows linearly with n. Another type of recursion, tree recursion, happens when the amount of information grows exponentially with the input. But we will leave it undiscussed here and go back shortly afterwards.

Iteration

A different perspective on computing factorials is by first multiplying 1 by 2, then multiplying the result by 3, then by 4, and so on until n. More formally, the program can use a counter that counts from 1 up to n and compute the product simultaneously until the counter exceeds n. Therefore the program can be written as:

Program 2:

```
int factorial (int n) {
   int product = 1;
   for(int i=2; i<n; i++) {
      product *= i;
   }
   return product;
}
```

This program, by contrast to program 2, does not build a chain of multiplication. At each step, the computer only need to keep track of the current values of the product and i. This type of program is called iteration, whose state can be summarized by a fixed number of variables, a fixed rule that describes how the variables should be updated, and an end test that specifies conditions under which the process should terminate. Same as recursion, when the time required grows linearly with the input, we call the iteration linear recursion.

Recursion vs Iteration

Compared the two processes, we can find that they seem almost same, especially in term of mathematical function. They both require a number of steps proportional to n to compute n!. On the other hand, when we consider the running processes of the two programs, they evolve quite differently.

In the iterative case, the program variables provide a complete description of the state. If we stopped the computation in the middle, to resume it only need to supply the computer with all variables. However, in the recursive process, information is maintained by the computer, therefore "hidden" to the program. This makes it almost impossible to resume the program after stopping it.

Tree recursion

As described above, tree recursion happens when the amount of information grows exponentially with the input. For instance, consider the sequence of Fibonacci num-

bers defined as follows:

$$\mathrm{Fib}(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & \text{otherwise} \end{cases}$$

By the definition, Fibonacci numbers have the following sequence, where each number is the sum of the previous two: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

A recursive program can be immediately written as:

Program 3:

```
int fib (int n) {
   if (n == 0) {
      return 0;
   } else if (n == 1) {
      return 1;
   } else {
      return fib(n-1) + fib(n-2);
   }
}
```

Therefore, to compute fib(5), the program computes fib(4) and fib(3). To computer fib(4), it computes fib(3) and fib(2). Notice that the fib procedure calls itself twice at the last line. Two observations can be obtained from the definition and the program:

- The ith Fibonacci number Fib(i) is equal to phi(i)/rootsquare(5) rounded to the nearest integer, which indicates that Fibonacci numbers grow exponentially.
- This is a bad way to compute Fibonacci numbers because it does redundant computation. Computing the running time of this procedure is beyond the scope of this article, but one can easily find that in books of algorithms, which is O(phi(n)). Thus, the program takes an amount of time that grows exponentiallywith the input.

On the other hand, we can also write the program in an iterative way for computing the Fibonacci numbers. Program 4 is a linear iteration. The difference in time required by Program 3 and 4 is enormous, even for small inputs.

Program 4:

```
int fib (int n) {
   int fib = 0;
   int a = 1;
   for(int i=0; i<n; i++) {
     fib = fib + a;
     a = fib;
   }
   return fib;
}
```

However, one should not think tree-recursive programs are useless. When we consider programs that operate on hierarchically data structures rather than numbers, tree-recursion is a natural and powerful tool. It can help us understand and design programs. Compared with Program 3 and 4, we can easily tell Program 3 is more straightforward, even if less efficient. After that, we can most likely reformulate the program into an iterative way.

90 Edit Distance in Java

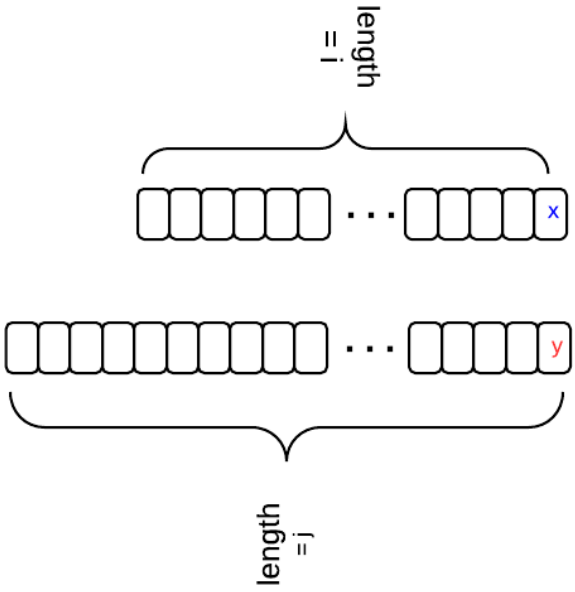
From Wiki:

> *In computer science, edit distance is a way of quantifying how dissimilar two strings (e.g., words) are to one another by counting the minimum number of operations required to transform one string into the other.*

There are three operations permitted on a word: replace, delete, insert. For example, the edit distance between "a" and "b" is 1, the edit distance between "abc" and "def" is 3. This post analyzes how to calculate edit distance by using dynamic programming.

Key Analysis

Let dp[i][j] stands for the edit distance between two strings with length i and j, i.e., word1[0,...,i-1] and word2[0,...,j-1]. There is a relation between dp[i][j] and dp[i-1][j-1]. Let's say we transform from one string to another. The first string has length i and it's last character is "x"; the second string has length j and its last character is "y". The following diagram shows the relation.



- if x == y, then dp[i][j] == dp[i-1][j-1]
- if x != y, and we insert y for word1, then dp[i][j] = dp[i][j-1] + 1
- if x != y, and we delete x for word1, then dp[i][j] = dp[i-1][j] + 1
- if x != y, and we replace x with y for word1, then dp[i][j] = dp[i-1][j-1] + 1
- When x!=y, dp[i][j] is the min of the three situations.

Initial condition: dp[i][0] = i, dp[0][j] = j

Java Code

After the analysis above, the code is just a representation of it.

```
public static int minDistance(String word1, String word2) {
  int len1 = word1.length();
  int len2 = word2.length();

  // len1+1, len2+1, because finally return dp[len1][len2]
  int[][] dp = new int[len1 + 1][len2 + 1];

  for (int i = 0; i <= len1; i++) {
    dp[i][0] = i;
  }

  for (int j = 0; j <= len2; j++) {
    dp[0][j] = j;
  }
```

```
  //iterate though, and check last char
  for (int i = 0; i < len1; i++) {
    char c1 = word1.charAt(i);
    for (int j = 0; j < len2; j++) {
      char c2 = word2.charAt(j);

      //if last two chars equal
      if (c1 == c2) {
        //update dp value for +1 length
        dp[i + 1][j + 1] = dp[i][j];
      } else {
        int replace = dp[i][j] + 1;
        int insert = dp[i][j + 1] + 1;
        int delete = dp[i + 1][j] + 1;

        int min = replace > insert ? insert : replace;
        min = delete > min ? min : delete;
        dp[i + 1][j + 1] = min;
      }
    }
  }

  return dp[len1][len2];
}
```

91 Single Number

The problem:

> *Given an array of integers, every element appears twice except for one. Find that single one.*

Thoughts

The key to solve this problem is bit manipulation. XOR will return 1 only on two different bits. So if two numbers are the same, XOR will return 0. Finally only one number left.

Java Solution

```
public class Solution {
   public int singleNumber(int[] A) {
```

```
        int x=0;

        for(int a: A){
            x = x ^ a;
        }

        return x;
    }
}
```

The question now is do you know any other ways to do this?

92 Single Number II

Problem

Given an array of integers, every element appears three times except for one. Find that single one.

Java Solution

This problem is similar to Single Number.

```
public int singleNumber(int[] A) {
    int ones = 0, twos = 0, threes = 0;
    for (int i = 0; i < A.length; i++) {
        twos |= ones & A[i];
        ones ^= A[i];
        threes = ones & twos;
        ones &= ~threes;
        twos &= ~threes;
    }
    return ones;
}
```

93 Twitter Codility Problem Max Binary Gap

Problem: Get maximum binary Gap.

For example, 9's binary form is 1001, the gap is 2.

Thoughts

The key to solve this problem is the fact that an integer x & 1 will get the last digit of the integer.

Java Solution

```
public class Solution {
  public static int solution(int N) {
    int max = 0;
    int count = -1;
    int r = 0;

    while (N > 0) {
      // get right most bit & shift right
      r = N & 1;
      N = N >> 1;

      if (0 == r && count >= 0) {
        count++;
      }

      if (1 == r) {
        max = count > max ? count : max;
        count = 0;
      }
    }

    return max;
  }

  public static void main(String[] args) {
    System.out.println(solution(9));
  }
}
```

94 Number of 1 Bits

Problem

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

For example, the 32-bit integer '11' has binary representation 00000000000000000000000000001011, so the function should return 3.

Java Solution

```
public int hammingWeight(int n) {
   int count = 0;
   for(int i=1; i<33; i++){
      if(getBit(n, i) == true){
         count++;
      }
   }
   return count;
}

public boolean getBit(int n, int i){
   return (n & (1 << i)) != 0;
}
```

95 Reverse Bits

Problem

Reverse bits of a given 32 bits unsigned integer.

For example, given input 43261596 (represented in binary as 00000010100101000001111010011100), return 964176192 (represented in binary as 00111001011110000010100101000000).

Follow up: If this function is called many times, how would you optimize it?

Related problem: Reverse Integer

Java Solution

```
public int reverseBits(int n) {
  for (int i = 0; i < 16; i++) {
    n = swapBits(n, i, 32 - i - 1);
  }
```

```
  return n;
}

public int swapBits(int n, int i, int j) {
  int a = (n >> i) & 1;
  int b = (n >> j) & 1;

  if ((a ^ b) != 0) {
    return n ^= (1 << i) | (1 << j);
  }

  return n;
}
```

96 Permutations

Given a collection of numbers, return all possible permutations.

```
For example,
[1,2,3] have the following permutations:
[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].
```

Java Solution 1

We can get all permutations by the following steps:

```
[1]
[2, 1]
[1, 2]
[3, 2, 1]
[2, 3, 1]
[2, 1, 3]
[3, 1, 2]
[1, 3, 2]
[1, 2, 3]
```

Loop through the array, in each iteration, a new number is added to different locations of results of previous iteration. Start from an empty List.

```
public ArrayList<ArrayList<Integer>> permute(int[] num) {
  ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

  //start from an empty list
```

```
  result.add(new  ArrayList<Integer>());

  for (int i = 0; i < num.length; i++) {
    //list of list in current iteration of the array num
    ArrayList<ArrayList<Integer>> current = new
        ArrayList<ArrayList<Integer>>();

    for (ArrayList<Integer> l : result) {
      // # of locations to insert is largest index + 1
      for (int j = 0; j < l.size()+1; j++) {
        // + add num[i] to different locations
        l.add(j, num[i]);

        ArrayList<Integer> temp = new ArrayList<Integer>(l);
        current.add(temp);

        //System.out.println(temp);

        // - remove num[i] add
        l.remove(j);
      }
    }

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

  return result;
}
```

Java Solution 2

We can also recursively solve this problem. Swap each element with each element after it.

```
public ArrayList<ArrayList<Integer>> permute(int[] num) {
  ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
  permute(num, 0, result);
  return result;
}

void permute(int[] num, int start, ArrayList<ArrayList<Integer>> result) {

  if (start >= num.length) {
    ArrayList<Integer> item = convertArrayToList(num);
    result.add(item);
  }

  for (int j = start; j <= num.length - 1; j++) {
```

```
      swap(num, start, j);
      permute(num, start + 1, result);
      swap(num, start, j);
    }
  }

  private ArrayList<Integer> convertArrayToList(int[] num) {
    ArrayList<Integer> item = new ArrayList<Integer>();
    for (int h = 0; h < num.length; h++) {
      item.add(num[h]);
    }
    return item;
  }

  private void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }
```

97 Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

```
For example, [1,1,2] have the following unique permutations:
[1,1,2], [1,2,1], and [2,1,1].
```

Basic Idea

For each number in the array, swap it with every element after it. To avoid duplicate, we need to check the existing sequence first.

Java Solution 1

```
public ArrayList<ArrayList<Integer>> permuteUnique(int[] num) {
  ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
  permuteUnique(num, 0, result);
  return result;
}
```

```
private void permuteUnique(int[] num, int start,
    ArrayList<ArrayList<Integer>> result) {

  if (start >= num.length ) {
    ArrayList<Integer> item = convertArrayToList(num);
    result.add(item);
  }

  for (int j = start; j <= num.length-1; j++) {
    if (containsDuplicate(num, start, j)) {
      swap(num, start, j);
      permuteUnique(num, start + 1, result);
      swap(num, start, j);
    }
  }
}

private ArrayList<Integer> convertArrayToList(int[] num) {
  ArrayList<Integer> item = new ArrayList<Integer>();
  for (int h = 0; h < num.length; h++) {
    item.add(num[h]);
  }
  return item;
}

private boolean containsDuplicate(int[] arr, int start, int end) {
  for (int i = start; i <= end-1; i++) {
    if (arr[i] == arr[end]) {
      return false;
    }
  }
  return true;
}

private void swap(int[] a, int i, int j) {
  int temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

Java Solution 2

Use set to maintain uniqueness:

```
public static ArrayList<ArrayList<Integer>> permuteUnique(int[] num) {
  ArrayList<ArrayList<Integer>> returnList = new
      ArrayList<ArrayList<Integer>>();
  returnList.add(new ArrayList<Integer>());
```

```
    for (int i = 0; i < num.length; i++) {
      Set<ArrayList<Integer>> currentSet = new HashSet<ArrayList<Integer>>();
      for (List<Integer> l : returnList) {
        for (int j = 0; j < l.size() + 1; j++) {
          l.add(j, num[i]);
          ArrayList<Integer> T = new ArrayList<Integer>(l);
          l.remove(j);
          currentSet.add(T);
        }
      }
      returnList = new ArrayList<ArrayList<Integer>>(currentSet);
    }

    return returnList;
}
```

Thanks to Milan for such a simple solution!

98 Permutation Sequence

The set [1,2,3,...,n] contains a total of n! unique permutations.

By listing and labeling all of the permutations in order, We get the following sequence (ie, for n = 3):

```
"123"
"132"
"213"
"231"
"312"
"321"
```

Given n and k, return the kth permutation sequence.

Note: Given n will be between 1 and 9 inclusive.

Thoughts

Naively loop through all cases will not work.

Java Solution 1

```
public class Solution {
  public String getPermutation(int n, int k) {

    // initialize all numbers
```

```
    ArrayList<Integer> numberList = new ArrayList<Integer>();
    for (int i = 1; i <= n; i++) {
      numberList.add(i);
    }

    // change k to be index
    k--;

    // set factorial of n
    int mod = 1;
    for (int i = 1; i <= n; i++) {
      mod = mod * i;
    }

    String result = "";

    // find sequence
    for (int i = 0; i < n; i++) {
      mod = mod / (n - i);
      // find the right number(curIndex) of
      int curIndex = k / mod;
      // update k
      k = k % mod;

      // get number according to curIndex
      result += numberList.get(curIndex);
      // remove from list
      numberList.remove(curIndex);
    }

    return result.toString();
  }
}
```

Java Solution 2

```
public class Solution {
  public String getPermutation(int n, int k) {
    boolean[] output = new boolean[n];
    StringBuilder buf = new StringBuilder("");

    int[] res = new int[n];
    res[0] = 1;

    for (int i = 1; i < n; i++)
      res[i] = res[i - 1] * i;

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

```
        int s = 1;

        while (k > res[i]) {
          s++;
          k = k - res[i];
        }

        for (int j = 0; j < n; j++) {
          if (j + 1 <= s && output[j]) {
            s++;
          }
        }

        output[s - 1] = true;
        buf.append(Integer.toString(s));
      }

      return buf.toString();
    }
}
```

99 Generate Parentheses

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given n = 3, a solution set is:

```
"((()))", "(()())", "(())()", "()(())", "()()()"
```

Java Solution

Read the following solution, give n=2, walk though the code. Hopefully you will quickly get an idea.

```
public List<String> generateParenthesis(int n) {
  ArrayList<String> result = new ArrayList<String>();
  ArrayList<Integer> diff = new ArrayList<Integer>();

  result.add("");
  diff.add(0);

  for (int i = 0; i < 2 * n; i++) {
    ArrayList<String> temp1 = new ArrayList<String>();
```

```
    ArrayList<Integer> temp2 = new ArrayList<Integer>();

    for (int j = 0; j < result.size(); j++) {
      String s = result.get(j);
      int k = diff.get(j);

      if (i < 2 * n - 1) {
        temp1.add(s + "(");
        temp2.add(k + 1);
      }

      if (k > 0 && i < 2 * n - 1 || k == 1 && i == 2 * n - 1) {
        temp1.add(s + ")");
        temp2.add(k - 1);
      }
    }

    result = new ArrayList<String>(temp1);
    diff = new ArrayList<Integer>(temp2);
  }

  return result;
}
```

Solution is provided first now. I will come back and draw a diagram to explain the solution.

100 Reverse Integer

LeetCode - Reverse Integer:

Reverse digits of an integer. Example1: x = 123, return 321 Example2: x = -123, return -321

Naive Method

We can convert the integer to a string/char array, reverse the order, and convert the string/char array back to an integer. However, this will require extra space for the string. It doesn't seem to be the right way, if you come with such a solution.

Efficient Approach

Actually, this can be done by using the following code.

```
public int reverse(int x) {
```

```
  //flag marks if x is negative
  boolean flag = false;
  if (x < 0) {
    x = 0 - x;
    flag = true;
  }

  int res = 0;
  int p = x;

  while (p > 0) {
    int mod = p % 10;
    p = p / 10;
    res = res * 10 + mod;
  }

  if (flag) {
    res = 0 - res;
  }

  return res;
}
```

Succinct Solution

This solution is from Sherry, it is succinct and it is pretty.

```
public int reverse(int x) {
   int rev = 0;
   while(x != 0){
      rev = rev*10 + x%10;
      x = x/10;
   }

   return rev;
}
```

Handle Out of Range Problem

As we form a new integer, it is possible that the number is out of range. We can use the following code to assign the newly formed integer. When it is out of range, throw an exception.

```
try{
  result = ...;
}catch(InputMismatchException exception){
  System.out.println("This is not an integer");
```

```
}
```

Please leave your comment if there is any better solutions.

101 Palindrome Number

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

Thoughts

Problems related with numbers are frequently solved by / and

Note: no extra space here means do not convert the integer to string, since string will be a copy of the integer and take extra space. The space take by div, left, and right can be ignored.

Java Solution

```
public class Solution {
   public boolean isPalindrome(int x) {
      //negative numbers are not palindrome
     if (x < 0)
       return false;

     // initialize how many zeros
     int div = 1;
     while (x / div >= 10) {
       div *= 10;
     }

     while (x != 0) {
       int left = x / div;
       int right = x % 10;

       if (left != right)
         return false;

       x = (x % div) / 10;
       div /= 100;
     }

     return true;
   }
}
```

102 Pow(x, n)

Problem:

Implement pow(x, n).

This is a great example to illustrate how to solve a problem during a technical interview. The first and second solution exceeds time limit; the third and fourth are accepted.

Naive Method

First of all, assuming n is not negative, to calculate x to the power of n, we can simply multiply x n times, i.e., x * x * ... * x. The time complexity is O(n). The implementation is as simple as:

```
public class Solution {
   public double pow(double x, int n) {
      if(x == 0) return 0;
      if(n == 0) return 1;

      double result=1;
      for(int i=1; i<=n; i++){
         result = result * x;
      }

      return result;
   }
}
```

Now we should think about how to do better than O(n).

Recursive Method

Naturally, we next may think how to do it in O(logn). We have a relation that xn̂ = x^(n/2) * x^(n/2) * x^(n

```
public static double pow(double x, int n) {
   if(n == 0)
      return 1;

   if(n == 1)
      return x;
```

```
    int half = n/2;
    int remainder = n%2;


    if(n % 2 ==1 && x < 0 && n < 0)
        return - 1/(pow(-x, half) * pow(-x, half) * pow(-x, remainder));
    else if (n < 0)
        return 1/(pow(x, -half) * pow(x, -half) * pow(x, -remainder));
    else
        return (pow(x, half) * pow(x, half) * pow(x, remainder));
}
```

In this solution, we can handle cases that x <0 and n <0. This solution actually takes more time than the first solution. Why?

Accepted Solution

The accepted solution is also recursive, but does division first. Time complexity is O(nlog(n)). The key part of solving this problem is the while loop.

```
public double pow(double x, int n) {
  if (n == 0)
    return 1;
  if (n == 1)
    return x;

  int pn = n > 0 ? n : -n;// positive n
  int pn2 = pn;

  double px = x > 0 ? x : -x;// positive x
  double result = px;

  int k = 1;
  //the key part of solving this problem
  while (pn / 2 > 0) {
    result = result * result;
    pn = pn / 2;
    k = k * 2;
  }

  result = result * pow(px, pn2 - k);

  // handle negative result
  if (x < 0 && n % 2 == 1)
    result = -result;

  // handle negative power
  if (n < 0)
    result = 1 / result;
```

```
  return result;
}
```

Best Solution

The most understandable solution I have found so far.

```
public double power(double x, int n) {
  if (n == 0)
    return 1;

  double v = power(x, n / 2);

  if (n % 2 == 0) {
    return v * v;
  } else {
    return v * v * x;
  }
}

public double pow(double x, int n) {
  if (n < 0) {
    return 1 / power(x, -n);
  } else {
    return power(x, n);
  }
}
```