Coding Interview in Java

121 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.

121.1 Simple Example

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

```java
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);
    }
    // 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]
```

121.2 Example of Solving Problems Using PriorityQueue

Merging k sorted list.

For more details about PriorityQueue, please go to doc.

122 Binary Tree Preorder Traversal

122.1 Analysis

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

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

122.2 Java Solution

```java
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;
    }
}
```

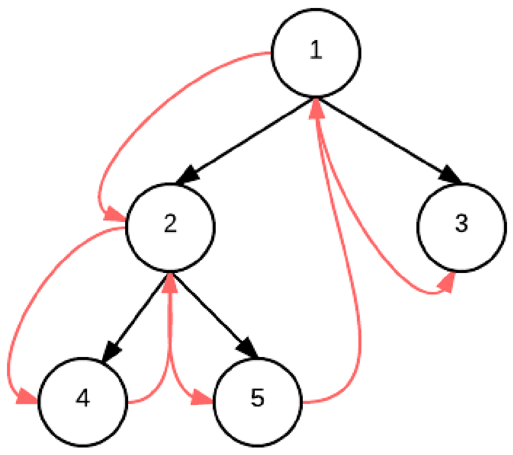
123 Binary Tree Inorder Traversal

There are 3 solutions for solving this problem.

123.1 Java Solution 1 - Iterative

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



```java
//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
```

```
         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;
     }
  }
```

123.2 Java Solution 2 - Recursive

The recursive solution is trivial.

```
public class Solution {
   List<Integer> result = new ArrayList<Integer>();

   public List<Integer> inorderTraversal(TreeNode root) {
      if(root !=null){
         helper(root);
      }

      return result;
   }

   public void helper(TreeNode p){
         helper(p.left);

      result.add(p.val);

      if(p.right!=null)
         helper(p.right);
   }
}
```

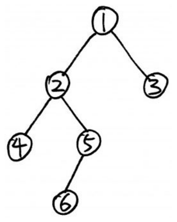
123.3 Java Solution 3 - Simple

Updated on 4/28/2016

```
public List<Integer> inorderTraversal(TreeNode root) {
   List<Integer> result = new ArrayList<Integer>();
   if(root==null)
      return result;
   Stack<TreeNode> stack = new Stack<TreeNode>();
   stack.push(root);

   while(!stack.isEmpty()){
      TreeNode top = stack.peek();
      if(top.left!=null){
         stack.push(top.left);
         top.left=null;
      }else{
         result.add(top.val);
         stack.pop();
         if(top.right!=null){
            stack.push(top.right);
         }
      }
   }

   return result;
}
```

124 Binary Tree Postorder Traversal

Among preoder, inorder and postorder binary tree traversal problems, postorder traversal is the most complicated one.



124.1 Java Solution 1

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 to the lft, check the previously visited node. If the current node is the left or right child of the previous node, then keep going down the tree, and add left/right node to stack when applicable. When there is no children for current node, i.e., the current node is a leaf, pop it from the stack. Then the previous node become to be under the current node for next loop. You can using an example to walk through the code.

```java
//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;
 }
```

124.2 Java Solution 2 - Simple!

Thanks to Edmond. This solution is superior!

```java
public List<Integer> postorderTraversal(TreeNode root) {
   List<Integer> res = new ArrayList<Integer>();

   if(root==null) {
      return res;
   }

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

   while(!stack.isEmpty()) {
      TreeNode temp = stack.peek();
      if(temp.left==null && temp.right==null) {
         TreeNode pop = stack.pop();
         res.add(pop.val);
      }
      else {
         if(temp.right!=null) {
            stack.push(temp.right);
            temp.right = null;
         }

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

   return res;
}
```

125 Binary Tree Level Order Traversal

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example: Given binary tree 3,9,20,#,#,15,7,

```
3
  / \
 9  20
   / \
  15  7
```

return its level order traversal as [[3], [9,20], [15,7]]

125.1 Analysis

It is obvious that this problem can be solve by using a queue. However, if we use one queue we can not track when each level starts. So we use two queues to track the current level and the next level.

125.2 Java Solution

```java
public ArrayList<ArrayList<Integer>> levelOrder(TreeNode root) {
   ArrayList<ArrayList<Integer>> al = new ArrayList<ArrayList<Integer>>();
   ArrayList<Integer> nodeValues = new ArrayList<Integer>();
   if(root == null)
      return al;

   LinkedList<TreeNode> current = new LinkedList<TreeNode>();
   LinkedList<TreeNode> next = new LinkedList<TreeNode>();
   current.add(root);

   while(!current.isEmpty()){
      TreeNode node = current.remove();

      if(node.left != null)
         next.add(node.left);
      if(node.right != null)
         next.add(node.right);
```

```
            if(current.isEmpty()){
               current = next;
               next = new LinkedList<TreeNode>();
               al.add(nodeValues);
               nodeValues = new ArrayList();
            }

      }
      return al;
   }
```

126 Binary Tree Level Order Traversal II

Given a binary tree, return the bottom-up level order traversal of its nodes' values.
For example, given binary tree 3,9,20,#,#,15,7,

```
3
  / \
 9  20
   / \
  15  7
```

return its level order traversal as [[15,7], [9,20],[3]]

126.1 Java Solution

```java
public List<ArrayList<Integer>> levelOrderBottom(TreeNode root) {
   ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

   if(root == null){
      return result;
   }

   LinkedList<TreeNode> current = new LinkedList<TreeNode>();
   LinkedList<TreeNode> next = new LinkedList<TreeNode>();
   current.offer(root);

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

   // need to track when each level starts
   while(!current.isEmpty()){
      TreeNode head = current.poll();

      numberList.add(head.val);

      if(head.left != null){
         next.offer(head.left);
      }
      if(head.right!= null){
         next.offer(head.right);
      }

      if(current.isEmpty()){
            next = new LinkedList<TreeNode>();
            result.add(numberList);
            numberList = new ArrayList<Integer>();
         }
      }

      //return Collections.reverse(result);
      ArrayList<ArrayList<Integer>> reversedResult = new
          ArrayList<ArrayList<Integer>>();
      for(int i=result.size()-1; i>=0; i--){
         reversedResult.add(result.get(i));
      }

      return reversedResult;
   }
```

127 Binary Tree Vertical Order Traversal

Given a binary tree, return the vertical order traversal of its nodes' values. (ie, from top to bottom, column by column).

127.1 Java Solution

For each node, its left child's degree is -1 and is right child's degree is +1. We can do a level order traversal and save the degree information.

```
class Wrapper{
   TreeNode node;
   int level;

   public Wrapper(TreeNode n, int l){
      node = n;
      level = l;
   }
}

public class Solution {
   public List<List<Integer>> verticalOrder(TreeNode root) {
      List<List<Integer>> result = new ArrayList<List<Integer>>();
      if(root==null)
         return result;

      LinkedList<Wrapper> queue = new LinkedList<Wrapper>();
      queue.offer(new Wrapper(root,0));

      TreeMap<Integer, ArrayList<Integer>> map = new TreeMap<Integer,
          ArrayList<Integer>>();

      while(!queue.isEmpty()){
         Wrapper w = queue.poll();

         TreeNode node = w.node;
         int level = w.level;

         if(map.containsKey(level)){
            map.get(level).add(node.val);
         }else{
            ArrayList<Integer> t = new ArrayList<Integer>();
            map.put(level, t);
         }

         if(node.left!=null){
            queue.offer(new Wrapper(node.left, level-1));
         }

         if(node.right!=null){
            queue.offer(new Wrapper(node.right, level+1));
         }

      }

      for(Map.Entry<Integer, ArrayList<Integer>> entry: map.entrySet()){
         result.add(entry.getValue());
      }

      return result;
   }
}
```

128 Invert Binary Tree

Google: 90

Very funny. Luckily, I can and in 2 ways!

128.1 Java Solution 1 - Recursive

```
public TreeNode invertTree(TreeNode root) {
   if(root!=null){
      helper(root);
   }

   return root;
}

public void helper(TreeNode p){

   TreeNode temp = p.left;
   p.left = p.right;
   p.right = temp;

   if(p.left!=null)
      helper(p.left);

   if(p.right!=null)
      helper(p.right);
}
```

128.2 Java Solution 2 - Iterative

```
public TreeNode invertTree(TreeNode root) {
   LinkedList<TreeNode> queue = new LinkedList<TreeNode>();

   if(root!=null){
      queue.add(root);
   }

   while(!queue.isEmpty()){
      TreeNode p = queue.poll();
```

```
            queue.add(p.left);
         if(p.right!=null)
            queue.add(p.right);

         TreeNode temp = p.left;
         p.left = p.right;
         p.right = temp;
      }

   return root;
}
```

129 Kth Smallest Element in a BST

Given a binary search tree, write a function kthSmallest to find the kth smallest element in it. ($1 \leq k \leq$ BST's total elements)

129.1 Java Solution 1 - Inorder Traversal

We can inorder traverse the tree and get the kth smallest element. Time is O(n).

```java
public int kthSmallest(TreeNode root, int k) {
   Stack<TreeNode> stack = new Stack<TreeNode>();

   TreeNode p = root;
   int result = 0;

   while(!stack.isEmpty() || p!=null){
      if(p!=null){
         stack.push(p);
         p = p.left;
      }else{
         TreeNode t = stack.pop();
         k--;
         if(k==0)
            result = t.val;
         p = t.right;
      }
   }

   return result;
}
```

129.2 Java Solution 2 - Extra Data Structure

We can let each node track the order, i.e., the number of elements that are less than itself. Time is O(log(n)).

coming soon...

130 Binary Tree Longest Consecutive Sequence

Given a binary tree, find the length of the longest consecutive sequence path.

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).

130.1 Java Solution 1 - Queue

```java
public int longestConsecutive(TreeNode root) {
   if(root==null)
      return 0;

   LinkedList<TreeNode> nodeQueue = new LinkedList<TreeNode>();
   LinkedList<Integer> sizeQueue = new LinkedList<Integer>();

   nodeQueue.offer(root);
   sizeQueue.offer(1);
   int max=1;

   while(!nodeQueue.isEmpty()){
      TreeNode head = nodeQueue.poll();
      int size = sizeQueue.poll();

      if(head.left!=null){
         int leftSize=size;
         if(head.val==head.left.val-1){
            leftSize++;
            max = Math.max(max, leftSize);
         }else{
            leftSize=1;
         }

         nodeQueue.offer(head.left);
         sizeQueue.offer(leftSize);
      }

      if(head.right!=null){
         int rightSize=size;
```

```
                rightSize++;
                max = Math.max(max, rightSize);
            }else{
                rightSize=1;
            }

            nodeQueue.offer(head.right);
            sizeQueue.offer(rightSize);
        }


    }

    return max;
}
```

130.2 Java Solution 2 - Recursion

```
public class Solution {
    int max=0;

    public int longestConsecutive(TreeNode root) {
        helper(root);
        return max;
    }

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

        int l = helper(root.left);
        int r = helper(root.right);

        int fromLeft = 0;
        int fromRight= 0;

        if(root.left==null){
            fromLeft=1;
        }else if(root.left.val-1==root.val){
            fromLeft = l+1;
        }else{
            fromLeft=1;
        }

        if(root.right==null){
            fromRight=1;
            fromRight = r+1;
        }else{
            fromRight=1;
        }

        max = Math.max(max, fromLeft);
        max = Math.max(max, fromRight);

        return Math.max(fromLeft, fromRight);
    }

}
```

131 Validate Binary Search Tree

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.

131.1 Java Solution 1 - Recursive

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. So we just check the boundaries for each node.

```
public boolean isValidBST(TreeNode root) {
   return isValidBST(root, Double.NEGATIVE_INFINITY,
       Double.POSITIVE_INFINITY);
}

public boolean isValidBST(TreeNode p, double min, double max){
   if(p==null)
      return true;

   if(p.val <= min || p.val >= max)
      return false;

   return isValidBST(p.left, min, p.val) && isValidBST(p.right, p.val, max);
}
```

This solution also goes to the left subtree first. If the violation occurs close to the root but on the right subtree, the method still cost O(n). The second solution below can handle violations close to root node faster.

131.2 Java Solution 2 - Iterative

```
public class Solution {
   public boolean isValidBST(TreeNode root) {
      if(root == null)
      LinkedList<BNode> queue = new LinkedList<BNode>();
      queue.add(new BNode(root, Double.NEGATIVE_INFINITY,
          Double.POSITIVE_INFINITY));
      while(!queue.isEmpty()){
         BNode b = queue.poll();
         if(b.n.val <= b.left || b.n.val >=b.right){
            return false;
         }
         if(b.n.left!=null){
            queue.offer(new BNode(b.n.left, b.left, b.n.val));
         }
         if(b.n.right!=null){
            queue.offer(new BNode(b.n.right, b.n.val, b.right));
         }
      }
      return true;
   }
}
//define a BNode class with TreeNode and it's boundaries
class BNode{
   TreeNode n;
   double left;
   double right;
   public BNode(TreeNode n, double left, double right){
      this.n = n;
      this.left = left;
      this.right = right;
   }
}
```

132 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
```

132.1 Thoughts

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

132.2 Java Solution

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

```
   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;
      }
   }
}
```

133 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.

133.1 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.

```java
/**
 * 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);
```

```
         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;
   }
}
```

133.2 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!

134 Path Sum II

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example, given the below binary tree and sum = 22,

```
5
         / \
        4   8
       /   / \
      11  13  4
     /  \    / \
    7    2  5   1
```

the method returns the following:

```
[
  [5,4,11,2],
  [5,8,4,5]
]
```

134.1 Analysis

This problem can be converted to be a typical depth-first search problem. A recursive depth-first search algorithm usually requires a recursive method call, a reference to the final result, a temporary result, etc.

134.2 Java Solution

```
public List<ArrayList<Integer>> pathSum(TreeNode root, int sum) {
   ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
   if(root == null)
      return result;

   ArrayList<Integer> l = new ArrayList<Integer>();
   l.add(root.val);
   dfs(root, sum-root.val, result, l);
   return result;
}
```

```
public void dfs(TreeNode t, int sum, ArrayList<ArrayList<Integer>> result,
    ArrayList<Integer> l){
   if(t.left==null && t.right==null && sum==0){
      ArrayList<Integer> temp = new ArrayList<Integer>();
      temp.addAll(l);
      result.add(temp);
   }

   //search path of left node
   if(t.left != null){
      l.add(t.left.val);
      dfs(t.left, sum-t.left.val, result, l);
      l.remove(l.size()-1);
   }

   //search path of right node
   if(t.right!=null){
      l.add(t.right.val);
      dfs(t.right, sum-t.right.val, result, l);
      l.remove(l.size()-1);
   }
}
```

135 Construct Binary Tree from Inorder and Postorder Traversal

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

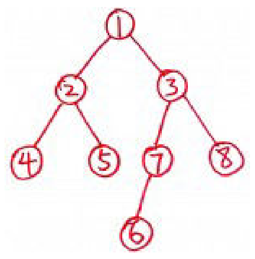
135.1 Analysis

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.



135.2 Java 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,
  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;
}
```

136 Construct Binary Tree from Preorder and Inorder Traversal

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

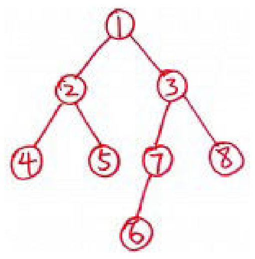
136.1 Analysis

Consider the following example:

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

From the pre-order array, we know that first 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 pre-order array. Recursively, we can build up the tree.



136.2 Java Solution

```
public TreeNode buildTree(int[] preorder, int[] inorder) {
   int preStart = 0;
   int preEnd = preorder.length-1;
   int inStart = 0;
   int inEnd = inorder.length-1;

   return construct(preorder, preStart, preEnd, inorder, inStart, inEnd);
}

public TreeNode construct(int[] preorder, int preStart, int preEnd, int[]
   if(preStart>preEnd||inStart>inEnd){
      return null;
   }

   int val = preorder[preStart];
   TreeNode p = new TreeNode(val);

   //find parent element index from inorder
   int k=0;
   for(int i=0; i<inorder.length; i++){
      if(val == inorder[i]){
         k=i;
         break;
      }
   }

   p.left = construct(preorder, preStart+1, preStart+(k-inStart), inorder,
       inStart, k-1);
   p.right= construct(preorder, preStart+(k-inStart)+1, preEnd, inorder, k+1
       , inEnd);

   return p;
}
```

137 Convert Sorted Array to Binary Search Tree

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

137.1 Thoughts

Straightforward! Recursively do the job.

137.2 Java Solution

```java
// 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);
    }
}
```

138 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.

138.1 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.

138.2 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 {
  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;
  }
}
```

139 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.

139.1 Thoughts

LinkedList is a queue in Java. The add() and remove() methods are used to manipulate the queue.

139.2 Java Solution

```java
/**
 * 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 && curr.right == null){
            return count;
         if(curr.left != null){
            nodes.add(curr.left);
            counts.add(count+1);
         }

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

      return 0;
   }
}
```

140 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
```

the result is 6.

140.1 Analysis

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

140.2 Java Solution

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

```
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;
}
```

141 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.

141.1 Analysis

This is a typical tree problem that can be solve by using recursion.

141.2 Java Solution

```java
// 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);
```

```
      return -1;

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

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

  }
}
```

142 Symmetric Tree

142.1 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
```

142.2 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;
  }
```

```
    return false;

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

  return true;
}
```

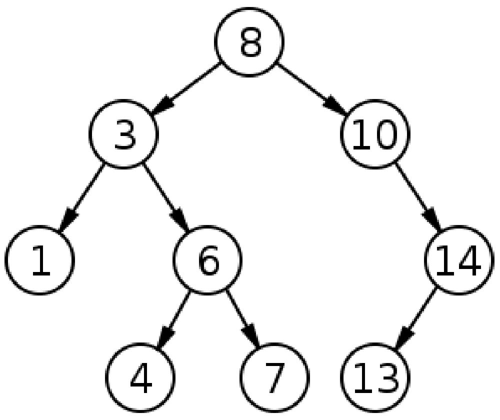
143 Binary Search Tree Iterator

143.1 Problem

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST. Calling next() will return the next smallest number in the BST. Note: next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

143.2 Java Solution

The key to solve this problem is understanding what is BST. Here is a diagram.



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

public class BSTIterator {
  Stack<TreeNode> stack;
```

```
    stack = new Stack<TreeNode>();
    while (root != null) {
      stack.push(root);
      root = root.left;
    }
  }

  public boolean hasNext() {
    return !stack.isEmpty();
  }

  public int next() {
    TreeNode node = stack.pop();
    int result = node.val;
    if (node.right != null) {
      node = node.right;
      while (node != null) {
        stack.push(node);
        node = node.left;
      }
    }
    return result;
  }
}
```

144 Binary Tree Right Side View

Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom. For example, given the following binary tree,

```
1            <---
 /   \
2    3          <---
 \
  5             <---
```

You can see [1, 3, 5].

144.1 Analysis

This problem can be solve by using a queue. On each level of the tree, we add the right-most element to the results.

144.2 Java Solution

```java
public List<Integer> rightSideView(TreeNode root) {
   ArrayList<Integer> result = new ArrayList<Integer>();

   if(root == null) return result;

   LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
   queue.add(root);

   while(queue.size() > 0){
      //get size here
      int size = queue.size();

      for(int i=0; i<size; i++){
         TreeNode top = queue.remove();

         //the first element in the queue (right-most of the tree)
         if(i==0){
            result.add(top.val);
         }
         //add right first
```

```
                queue.add(top.right);
            }
            //add left
            if(top.left != null){
                queue.add(top.left);
            }
        }
    }

    return result;
}
```

145 Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

145.1 Analysis

This problem can be solved by using BST property, i.e., left <parent <right for each node. There are 3 cases to handle.

145.2 Java Solution

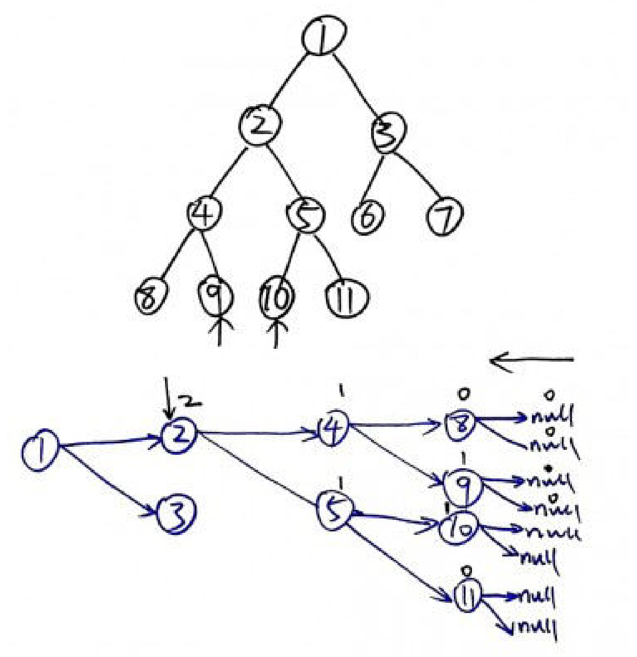
```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
   TreeNode m = root;

   if(m.val > p.val && m.val < q.val){
      return m;
   }else if(m.val>p.val && m.val > q.val){
      return lowestCommonAncestor(root.left, p, q);
   }else if(m.val<p.val && m.val < q.val){
      return lowestCommonAncestor(root.right, p, q);
   }

   return root;
}
```

146 Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

146.1 Java Solution 1

Please use the following diagram to walk through the solution.



Since each node is visited in the worst case, time complexity is O(n).

```java
class Entity{
   public int count;
   public TreeNode node;

   public Entity(int count, TreeNode node){
      this.count = count;
      this.node = node;
}

public class Solution {
   public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode
       q) {
      return lcaHelper(root, p, q).node;
   }

   public Entity lcaHelper(TreeNode root, TreeNode p, TreeNode q){
      if(root == null) return new Entity(0, null);

      Entity left = lcaHelper(root.left, p, q);
      if(left.count==2)
         return left;

      Entity right = lcaHelper(root.right,p,q);
      if(right.count==2)
         return right;

      int numTotal = left.count + right.count;
      if(root== p || root == q){
         numTotal++;
      }

      return new Entity(numTotal, root);
   }
}
```

146.2 Java Solution 2

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
   if(root == null || root == p || root == q) return root;
   TreeNode left = lowestCommonAncestor(root.left, p, q);
   TreeNode right = lowestCommonAncestor(root.right, p, q);
   if(left!=null&&right!=null) return root;
   return left == null ? right : left;
}
```

147 Verify Preorder Serialization of a Binary Tree

One way to serialize a binary tree is to use pre-order traversal. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as #.

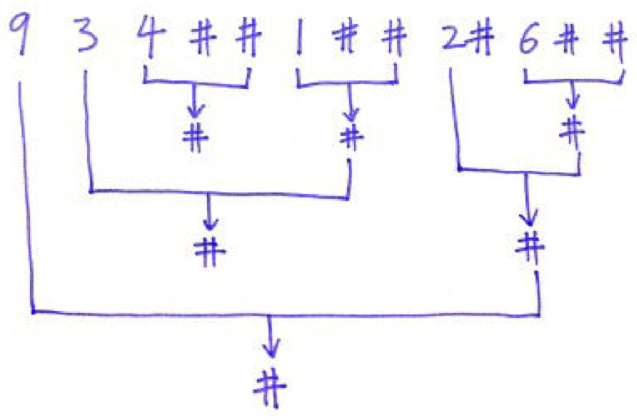
```
9
    /   \
   3     2
  / \   / \
 4   1  #  6
/ \ / \   / \
# # # #   # #
```

For example, the above binary tree can be serialized to the string "9,3,4,#,#,1,#,#,2,#,6,#,#", where # represents a null node.

Given a string of comma separated values, verify whether it is a correct preorder traversal serialization of a binary tree. Find an algorithm without reconstructing the tree.

147.1 Java Solution - Stack

We can keep removing the leaf node until there is no one to remove. If a sequence is like "4 # #", change it to "#" and continue. We need a stack so that we can record previous removed nodes.



```
public boolean isValidSerialization(String preorder) {
   LinkedList<String> stack = new LinkedList<String>();
   String[] arr = preorder.split(",");

   for(int i=0; i<arr.length; i++){
      stack.add(arr[i]);

      while(stack.size()>=3
         && stack.get(stack.size()-1).equals("#")
         && stack.get(stack.size()-2).equals("#")
         && !stack.get(stack.size()-3).equals("#")){

         stack.remove(stack.size()-1);
         stack.remove(stack.size()-1);
         stack.remove(stack.size()-1);

         stack.add("#");
      }

   }

   if(stack.size()==1 && stack.get(0).equals("#"))
      return true;
   else
      return false;
}
```

148 Populating Next Right Pointers in Each Node

Given the following perfect binary tree,

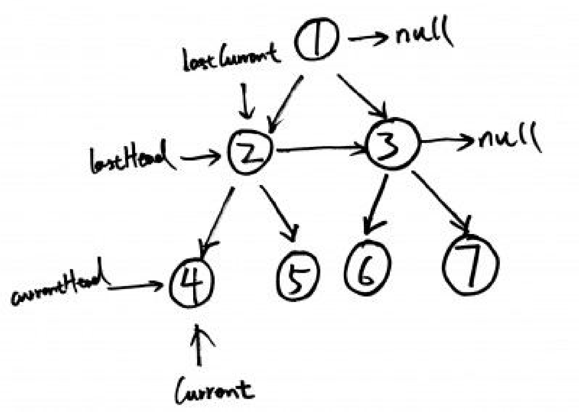
```
1
       / \
      2   3
     / \ / \
    4  5  6  7
```

After calling your function, the tree should look like:

```
1 -> NULL
       / \
      2 -> 3 -> NULL
     / \ / \
    4->5->6->7 -> NULL
```

148.1 Java Solution

This solution is easier to understand. You can use the example tree above to walk through the algorithm. The basic idea is have 4 pointers to move towards right on two levels (see comments in the code).



```
    if(root == null)
        return;

    TreeLinkNode lastHead = root;//prevous level’s head
    TreeLinkNode lastCurrent = null;//previous level’s pointer
    TreeLinkNode currentHead = null;//currnet level’s head
    TreeLinkNode current = null;//current level’s pointer

    while(lastHead!=null){
        lastCurrent = lastHead;

        while(lastCurrent!=null){
            if(currentHead == null){
                currentHead = lastCurrent.left;
                current = lastCurrent.left;
            }else{
                current.next = lastCurrent.left;
                current = current.next;
            }

            if(currentHead != null){
                current.next = lastCurrent.right;
                current = current.next;
            }

            lastCurrent = lastCurrent.next;
        }

        //update last head
        lastHead = currentHead;
        currentHead = null;
    }

}
```

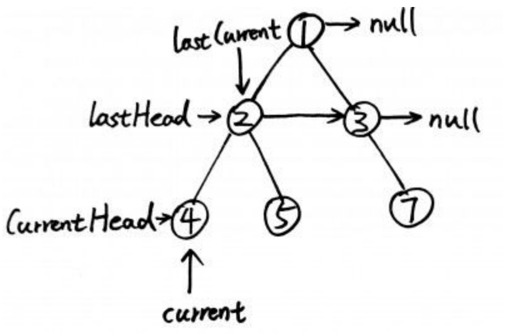
149 Populating Next Right Pointers in Each Node II

Follow up for problem "Populating Next Right Pointers in Each Node".

What if the given tree could be any binary tree? Would your previous solution still work?

149.1 Analysis

Similar to Populating Next Right Pointers in Each Node, we have 4 pointers at 2 levels of the tree.



149.2 Java Solution

```java
public void connect(TreeLinkNode root) {
   if(root == null)
      return;

   TreeLinkNode lastHead = root;//prevous level's head
   TreeLinkNode lastCurrent = null;//previous level's pointer
   TreeLinkNode currentHead = null;//currnet level's head
   TreeLinkNode current = null;//current level's pointer

   while(lastHead!=null){
      lastCurrent = lastHead;

      while(lastCurrent!=null){
         //left child is not null
```

```
            if(currentHead == null){
               currentHead = lastCurrent.left;
               current = lastCurrent.left;
            }else{
               current.next = lastCurrent.left;
               current = current.next;
            }
         }

         //right child is not null
         if(lastCurrent.right!=null){
            if(currentHead == null){
               currentHead = lastCurrent.right;
               current = lastCurrent.right;
            }else{
               current.next = lastCurrent.right;
               current = current.next;
            }
         }

         lastCurrent = lastCurrent.next;
      }

      //update last head
      lastHead = currentHead;
      currentHead = null;
   }
}
```

150 Unique Binary Search Trees

Given n, how many structurally unique BST's (binary search trees) that store values 1...n?

For example, Given n = 3, there are a total of 5 unique BST's.

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

150.1 Analysis

Let count[i] be the number of unique binary search trees for i. The number of trees are determined by the number of subtrees which have different root node. For example,

```
i=0, count[0]=1 //empty tree

i=1, count[1]=1 //one tree

i=2, count[2]=count[0]*count[1] // 0 is root
         + count[1]*count[0] // 1 is root

i=3, count[3]=count[0]*count[2] // 1 is root
         + count[1]*count[1] // 2 is root
         + count[2]*count[0] // 3 is root

i=4, count[4]=count[0]*count[3] // 1 is root
         + count[1]*count[2] // 2 is root
         + count[2]*count[1] // 3 is root
         + count[3]*count[0] // 4 is root
..
..
..

i=n, count[n] = sum(count[0..k]*count[k+1...n]) 0 <= k < n-1
```

Use dynamic programming to solve the problem.

150.2 Java Solution

```
public int numTrees(int n) {
  int[] count = new int[n + 1];

  count[0] = 1;
  count[1] = 1;

  for (int i = 2; i <= n; i++) {
    for (int j = 0; j <= i - 1; j++) {
      count[i] = count[i] + count[j] * count[i - j - 1];
    }
  }

  return count[n];
}
```

Check out how to get all unique binary search trees.

151 Unique Binary Search Trees II

Given n, generate all structurally unique BST's (binary search trees) that store values 1...n.

For example, Given n = 3, your program should return all 5 unique BST's shown below.

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

151.1 Analysis

Check out Unique Binary Search Trees I.

This problem can be solved by recursively forming left and right subtrees. The different combinations of left and right subtrees form the set of all unique binary search trees.

151.2 Java Solution

```java
public List<TreeNode> generateTrees(int n) {
   return generateTrees(1, n);
}

public List<TreeNode> generateTrees(int start, int end) {
   List<TreeNode> list = new LinkedList<>();

   if (start > end) {
      list.add(null);
      return list;
   }

   for (int i = start; i <= end; i++) {
      List<TreeNode> lefts = generateTrees(start, i - 1);
      List<TreeNode> rights = generateTrees(i + 1, end);
      for (TreeNode left : lefts) {
         for (TreeNode right : rights) {
            node.left = left;
            node.right = right;
            list.add(node);
         }
      }
   }

   return list;
}
```

152 Sum Root to Leaf Numbers

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number. Find the total sum of all root-to-leaf numbers.

For example,

```
1
  / \
 2   3
```

The root-to-leaf path 1->2 represents the number 12. The root-to-leaf path 1->3 represents the number 13. Return the sum = 12 + 13 = 25.

152.1 Java Solution - Recursive

This problem can be solved by a typical DFS approach.

```
public int sumNumbers(TreeNode root) {
   int result = 0;
   if(root==null)
      return result;

   ArrayList<ArrayList<TreeNode>> all = new ArrayList<ArrayList<TreeNode>>();
   ArrayList<TreeNode> l = new ArrayList<TreeNode>();
   l.add(root);
   dfs(root, l, all);

   for(ArrayList<TreeNode> a: all){
      StringBuilder sb = new StringBuilder();
      for(TreeNode n: a){
         sb.append(String.valueOf(n.val));
      }
      int currValue = Integer.valueOf(sb.toString());
      result = result + currValue;
   }

   return result;
}

public void dfs(TreeNode n, ArrayList<TreeNode> l,
    ArrayList<ArrayList<TreeNode>> all){
   if(n.left==null && n.right==null){
      t.addAll(l);
      all.add(t);
   }

   if(n.left!=null){
      l.add(n.left);
      dfs(n.left, l, all);
      l.remove(l.size()-1);
   }

   if(n.right!=null){
      l.add(n.right);
      dfs(n.right, l, all);
      l.remove(l.size()-1);
   }

}
```

Same approach, but simpler coding style.

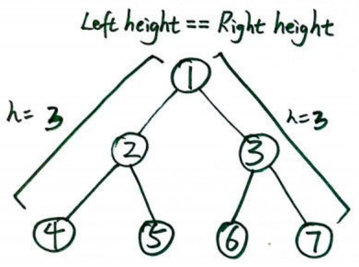
```
public int sumNumbers(TreeNode root) {
   if(root == null)
      return 0;

   return dfs(root, 0, 0);
}

public int dfs(TreeNode node, int num, int sum){
   if(node == null) return sum;

   num = num*10 + node.val;

   // leaf
   if(node.left == null && node.right == null) {
      sum += num;
      return sum;
   }

   // left subtree + right subtree
   sum = dfs(node.left, num, sum) + dfs(node.right, num, sum);
   return sum;
}
```

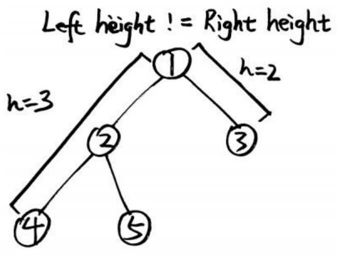
153 Count Complete Tree Nodes

Given a complete binary tree, count the number of nodes.

153.1 Analysis

Steps to solve this problem: 1) get the height of left-most part 2) get the height of right-most part 3) when they are equal, the # of nodes = 2ĥ -1 4) when they are not equal, recursively get # of nodes from left&right sub-trees





Time complexity is O(h2̂).

153.2 Java Solution

```java
public int countNodes(TreeNode root) {
   if(root==null)
      return 0;

   int left = getLeftHeight(root)+1;
   int right = getRightHeight(root)+1;
   if(left==right){
      return (2<<(left-1))-1;
   }else{
      return countNodes(root.left)+countNodes(root.right)+1;
   }
}

public int getLeftHeight(TreeNode n){
   if(n==null) return 0;

   int height=0;
   while(n.left!=null){
      height++;
      n = n.left;
   }
   return height;
}

public int getRightHeight(TreeNode n){
   if(n==null) return 0;

   int height=0;
   while(n.right!=null){
      height++;
      n = n.right;
   }
   return height;
}
```

154 Closest Binary Search Tree Value

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

154.1 Java Solution

Recursively traverse down the root. When target is less than root, go left; when target is greater than root, go right.

```java
public class Solution {
   int goal;
   double min = Double.MAX_VALUE;

   public int closestValue(TreeNode root, double target) {
      helper(root, target);
      return goal;
   }

   public void helper(TreeNode root, double target){
      if(root==null)
         return;

      if(Math.abs(root.val - target) < min){
         min = Math.abs(root.val-target);
         goal = root.val;
      }

      if(target < root.val){
         helper(root.left, target);
      }else{
         helper(root.right, target);
      }
   }
}
```

155 Binary Tree Paths

Given a binary tree, return all root-to-leaf paths.

155.1 Java Solution

A typical depth-first search problem.

```java
public List<String> binaryTreePaths(TreeNode root) {
   ArrayList<String> finalResult = new ArrayList<String>();

   if(root==null)
      return finalResult;

   ArrayList<String> curr = new ArrayList<String>();
   ArrayList<ArrayList<String>> results = new ArrayList<ArrayList<String>>();

   dfs(root, results, curr);

   for(ArrayList<String> al : results){
      StringBuilder sb = new StringBuilder();
      sb.append(al.get(0));
      for(int i=1; i<al.size();i++){
         sb.append("->"+al.get(i));
      }

      finalResult.add(sb.toString());
   }

   return finalResult;
}

public void dfs(TreeNode root, ArrayList<ArrayList<String>> list,
    ArrayList<String> curr){
   curr.add(String.valueOf(root.val));

   if(root.left==null && root.right==null){
      list.add(curr);
      return;
   }

   if(root.left!=null){
      ArrayList<String> temp = new ArrayList<String>(curr);
    }

    if(root.right!=null){
        ArrayList<String> temp = new ArrayList<String>(curr);
        dfs(root.right, list, temp);
    }
}
```

156 Merge K Sorted Arrays in Java

This is a classic interview question. Another similar problem is "merge k sorted lists".

This problem can be solved by using a heap. The time is O(nlog(n)).

Given m arrays, the minimum elements of all arrays can form a heap. It takes O(log(m)) to insert an element to the heap and it takes O(1) to delete the minimum element.

```
class ArrayContainer implements Comparable<ArrayContainer> {
  int[] arr;
  int index;

  public ArrayContainer(int[] arr, int index) {
    this.arr = arr;
    this.index = index;
  }

  @Override
  public int compareTo(ArrayContainer o) {
    return this.arr[this.index] - o.arr[o.index];
  }
}
```

```
public class KSortedArray {
  public static int[] mergeKSortedArray(int[][] arr) {
    //PriorityQueue is heap in Java
    PriorityQueue<ArrayContainer> queue = new PriorityQueue<ArrayContainer>();
    int total=0;

    //add arrays to heap
    for (int i = 0; i < arr.length; i++) {
      queue.add(new ArrayContainer(arr[i], 0));
      total = total + arr[i].length;
    }

    int m=0;
    int result[] = new int[total];

    //while heap is not empty
    while(!queue.isEmpty()){
      ArrayContainer ac = queue.poll();
      result[m++]=ac.arr[ac.index];
```

```
            if(ac.index < ac.arr.length-1){
                queue.add(new ArrayContainer(ac.arr, ac.index+1));
            }
        }

        return result;
    }

    public static void main(String[] args) {
        int[] arr1 = { 1, 3, 5, 7 };
        int[] arr2 = { 2, 4, 6, 8 };
        int[] arr3 = { 0, 9, 10, 11 };

        int[] result = mergeKSortedArray(new int[][] { arr1, arr2, arr3 });
        System.out.println(Arrays.toString(result));
    }
}
```

157 Merge k Sorted Lists

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

157.1 Analysis

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).

157.2 Java Solution

```java
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)
```

```
                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.

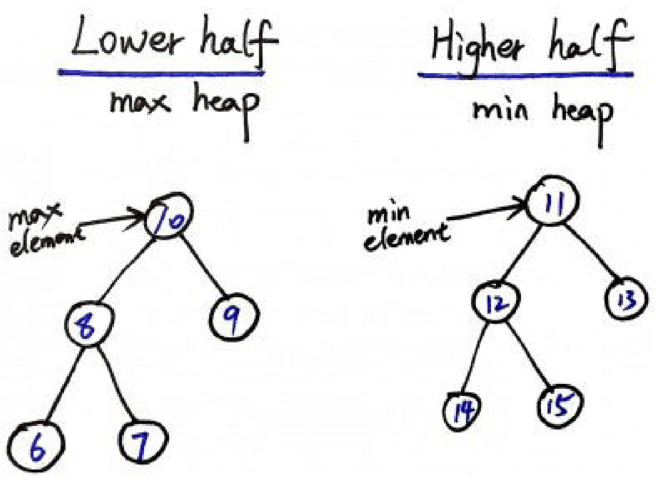
158 Find Median from Data Stream

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

158.1 Analysis

First of all, it seems that the best time complexity we can get for this problem is O(log(n)) of add() and O(1) of getMedian(). This data structure seems highly likely to be a tree.

We can use heap to solve this problem. In Java, the PriorityQueue class is a priority heap. We can use two heaps to store the lower half and the higher half of the data stream. The size of the two heaps differs at most 1.



158.2 Java Solution

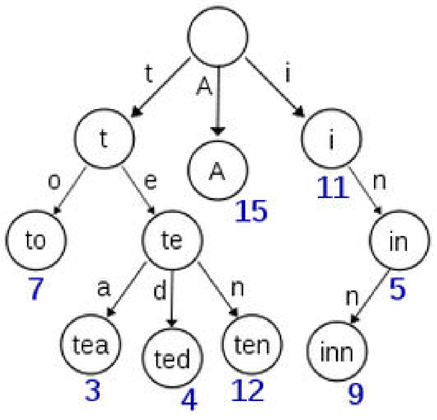
```
class MedianFinder {
   PriorityQueue<Integer> maxHeap;//lower half
   PriorityQueue<Integer> minHeap;//higher half

   public MedianFinder(){
      maxHeap = new PriorityQueue<Integer>(Collections.reverseOrder());
      minHeap = new PriorityQueue<Integer>();
   // Adds a number into the data structure.
   public void addNum(int num) {
      maxHeap.offer(num);
      minHeap.offer(maxHeap.poll());

      if(maxHeap.size() < minHeap.size()){
         maxHeap.offer(minHeap.poll());
      }
   }

   // Returns the median of current data stream
   public double findMedian() {
      if(maxHeap.size()==minHeap.size()){
         return (double)(maxHeap.peek()+(minHeap.peek()))/2;
      }else{
         return maxHeap.peek();
      }
   }
}
```

159 Implement Trie (Prefix Tree)

Implement a trie with insert, search, and startsWith methods.

159.1 Java Solution 1

A trie node should contains the character, its children and the flag that marks if it is a leaf node. You can use this diagram to walk though the Java solution.



```
class TrieNode {
   char c;
   HashMap<Character, TrieNode> children = new HashMap<Character, TrieNode>();
   boolean isLeaf;

   public TrieNode() {}

   public TrieNode(char c){
      this.c = c;
   }
}
```

```
public class Trie {
   private TrieNode root;

   public Trie() {
      root = new TrieNode();
   // Inserts a word into the trie.
   public void insert(String word) {
      HashMap<Character, TrieNode> children = root.children;

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

         TrieNode t;
         if(children.containsKey(c)){
               t = children.get(c);
         }else{
            t = new TrieNode(c);
            children.put(c, t);
         }

         children = t.children;

         //set leaf node
         if(i==word.length()-1)
            t.isLeaf = true;
      }
   }

   // Returns if the word is in the trie.
   public boolean search(String word) {
      TrieNode t = searchNode(word);

      if(t != null && t.isLeaf)
         return true;
      else
         return false;
   }

   // Returns if there is any word in the trie
   // that starts with the given prefix.
   public boolean startsWith(String prefix) {
      if(searchNode(prefix) == null)
         return false;
      else
         return true;
   }

   public TrieNode searchNode(String str){
      Map<Character, TrieNode> children = root.children;
      TrieNode t = null;
      for(int i=0; i<str.length(); i++){
         char c = str.charAt(i);
         if(children.containsKey(c)){
            children = t.children;
         }else{
            return null;
         }
      }

      return t;
   }
}
```

159.2 Java Solution 2 - Improve Performance by Using an Array

Each trie node can only contains 'a'-'z' characters. So we can use a small array to store the character.

```
class TrieNode {
   TrieNode[] arr;
   boolean isEnd;
   // Initialize your data structure here.
   public TrieNode() {
      this.arr = new TrieNode[26];
   }

}

public class Trie {
   private TrieNode root;

   public Trie() {
      root = new TrieNode();
   }

   // Inserts a word into the trie.
   public void insert(String word) {
      TrieNode p = root;
      for(int i=0; i<word.length(); i++){
         char c = word.charAt(i);
         int index = c-'a';
         if(p.arr[index]==null){
            TrieNode temp = new TrieNode();
            p.arr[index]=temp;
            p = temp;
         }else{
            p=p.arr[index];
         }
      }
   }

   // Returns if the word is in the trie.
   public boolean search(String word) {
      TrieNode p = searchNode(word);
      if(p==null){
         return false;
      }else{
         if(p.isEnd)
            return true;
      }

      return false;
   }

   // Returns if there is any word in the trie
   // that starts with the given prefix.
   public boolean startsWith(String prefix) {
      TrieNode p = searchNode(prefix);
      if(p==null){
         return false;
      }else{
         return true;
      }
   }

   public TrieNode searchNode(String s){
      TrieNode p = root;
      for(int i=0; i<s.length(); i++){
         char c= s.charAt(i);
         int index = c-'a';
         if(p.arr[index]!=null){
            p = p.arr[index];
         }else{
            return null;
         }
      }

      if(p==root)
         return null;

      return p;
   }
}
```

160 Add and Search Word Data structure design

Design a data structure that supports the following two operations:

```
void addWord(word)
bool search(word)
```

search(word) can search a literal word or a regular expression string containing only letters a-z or .. A . means it can represent any one letter.

160.1 Java Solution 1

This problem is similar with Implement Trie. The solution 1 below uses the same definition of a trie node. To handle the "." case for this problem, we need to search all possible paths, instead of one path.

TrieNode

```
class TrieNode{
   char c;
   HashMap<Character, TrieNode> children = new HashMap<Character, TrieNode>();
   boolean isLeaf;

   public TrieNode() {}

   public TrieNode(char c){
      this.c = c;
   }
}
```

WordDictionary

```
public class WordDictionary {
   private TrieNode root;

   public WordDictionary(){
      root = new TrieNode();
   }

   // Adds a word into the data structure.
   public void addWord(String word) {
      for(int i=0; i<word.length(); i++){
         char c = word.charAt(i);

         TrieNode t = null;
         if(children.containsKey(c)){
            t = children.get(c);
         }else{
            t = new TrieNode(c);
            children.put(c,t);
         }

         children = t.children;

         if(i == word.length()-1){
            t.isLeaf = true;
         }
      }
   }

   // Returns if the word is in the data structure. A word could
   // contain the dot character '.' to represent any one letter.
   public boolean search(String word) {
     return dfsSearch(root.children, word, 0);

   }

    public boolean dfsSearch(HashMap<Character, TrieNode> children, String
        word, int start) {
      if(start == word.length()){
         if(children.size()==0)
            return true;
         else
            return false;
      }

      char c = word.charAt(start);

      if(children.containsKey(c)){
         if(start == word.length()-1 && children.get(c).isLeaf){
            return true;
         }

         return dfsSearch(children.get(c).children, word, start+1);
      }else if(c == '.'){
         boolean result = false;
         for(Map.Entry<Character, TrieNode> child: children.entrySet()){
            if(start == word.length()-1 && child.getValue().isLeaf){
               return true;
            //if any path is true, set result to be true;
            if(dfsSearch(child.getValue().children, word, start+1)){
               result = true;
            }
         }

         return result;
      }else{
         return false;
      }
   }
}
```

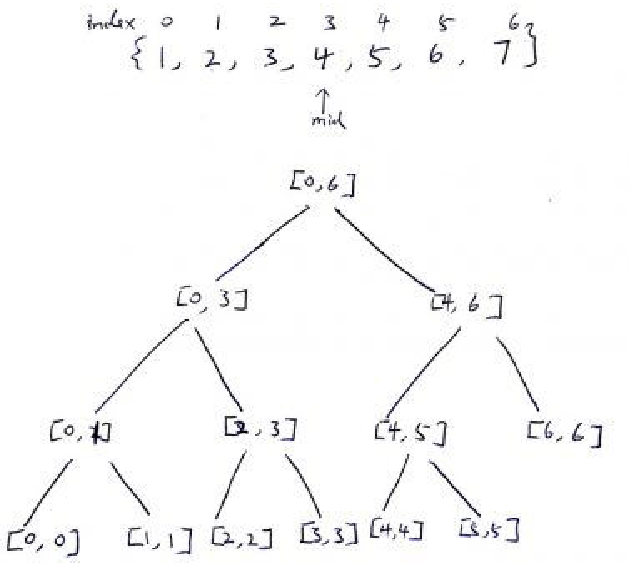
160.2 Java Solution 2 - Using Array Instead of HashMap

```
class TrieNode{
   TrieNode[] arr;
   boolean isLeaf;

   public TrieNode(){
      arr = new TrieNode[26];
   }
}

public class WordDictionary {
   TrieNode root;

   public WordDictionary(){
      root = new TrieNode();
   }
   // Adds a word into the data structure.
   public void addWord(String word) {
      TrieNode p= root;
      for(int i=0; i<word.length(); i++){
         char c=word.charAt(i);
         int index = c-'a';
         if(p.arr[index]==null){
            TrieNode temp = new TrieNode();
            p.arr[index]=temp;
            p=temp;
         }else{
            p=p.arr[index];
         }
      }
   }

   // Returns if the word is in the data structure. A word could
   // contain the dot character '.' to represent any one letter.
   public boolean search(String word) {
      return dfsSearch(root, word, 0);
   }

  public boolean dfsSearch(TrieNode p, String word, int start) {
    if (p.isLeaf && start == word.length())
      return true;

    if (start >= word.length())
      return false;

    char c = word.charAt(start);

    if (c == '.') {
      boolean tResult = false;
      for (int j = 0; j < 26; j++) {
        if (p.arr[j] != null) {
          if (dfsSearch(p.arr[j], word, start + 1)) {
            tResult = true;
            break;
          }
        }
      }

      if (tResult)
        return true;
    } else {
      int index = c - 'a';

      if (p.arr[index] != null) {
        return dfsSearch(p.arr[index], word, start + 1);
      } else {
        return false;
      }
    }

    return false;
  }
}
```

161 Range Sum Query Mutable

Given an integer array nums, find the sum of the elements between indices i and j (i $\leq$ j), inclusive. The update(i, val) function modifies nums by updating the element at index i to val.

161.1 Java Solution



```
class TreeNode{
   int start;
   int end;
   int sum;
   TreeNode leftChild;
   TreeNode rightChild;

   public TreeNode(int left, int right, int sum){
      this.start=left;
      this.end=right;
      this.sum=sum;
   }
   public TreeNode(int left, int right){
      this.end=right;
      this.sum=0;
   }
}

public class NumArray {
   TreeNode root = null;

   public NumArray(int[] nums) {
      if(nums==null || nums.length==0)
         return;

      this.root = buildTree(nums, 0, nums.length-1);
   }

   void update(int i, int val) {
      updateHelper(root, i, val);
   }

   void updateHelper(TreeNode root, int i, int val){
      if(root==null)
         return;



      int mid = root.start + (root.end-root.start)/2;
      if(i<=mid){
         updateHelper(root.leftChild, i, val);
      }else{
         updateHelper(root.rightChild, i, val);
      }

      if(root.start==root.end&& root.start==i){
         root.sum=val;
         return;
      }

      root.sum=root.leftChild.sum+root.rightChild.sum;
   }

   public int sumRange(int i, int j) {
      return sumRangeHelper(root, i, j);
   }

   public int sumRangeHelper(TreeNode root, int i, int j){
      if(root==null || j<root.start || i > root.end ||i>j )
         return 0;

      if(i<=root.start && j>=root.end){
      }
      int mid = root.start + (root.end-root.start)/2;
      int result = sumRangeHelper(root.leftChild, i, Math.min(mid, j))
               +sumRangeHelper(root.rightChild, Math.max(mid+1, i), j);

      return result;
   }

   public TreeNode buildTree(int[] nums, int i, int j){
      if(nums==null || nums.length==0|| i>j)
         return null;

      if(i==j){
         return new TreeNode(i, j, nums[i]);
      }

      TreeNode current = new TreeNode(i, j);

      int mid = i + (j-i)/2;

      current.leftChild = buildTree(nums, i, mid);
      current.rightChild = buildTree(nums, mid+1, j);

      current.sum = current.leftChild.sum+current.rightChild.sum;

      return current;
   }
}
```

162 The Skyline Problem

162.1 Analysis

This problem is essentially a problem of processing 2*n edges. Each edge has a x-axis value and a height value. The key part is how to use the height heap to process each edge.

162.2 Java Solution

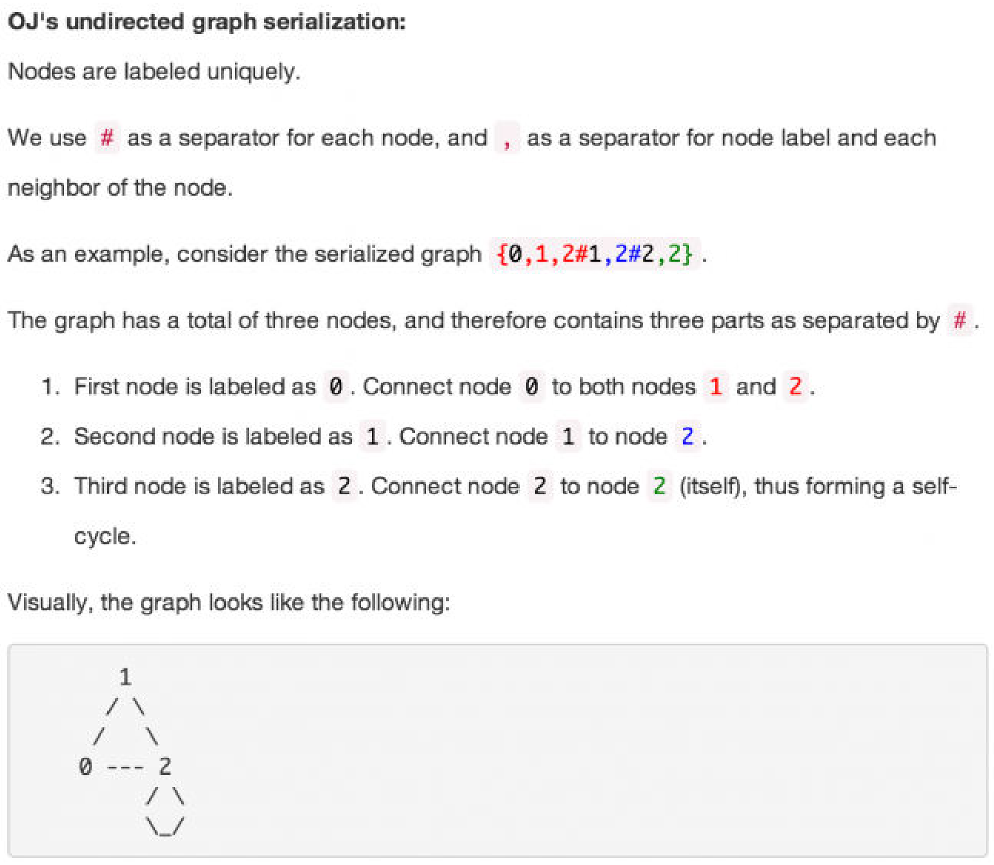
```java
class Edge {
  int x;
  int height;
  boolean isStart;

  public Edge(int x, int height, boolean isStart) {
    this.x = x;
    this.height = height;
    this.isStart = isStart;
  }
}
```

```java
public List<int[]> getSkyline(int[][] buildings) {
  List<int[]> result = new ArrayList<int[]>();

  if (buildings == null || buildings.length == 0
      || buildings[0].length == 0) {
    return result;
  }

  List<Edge> edges = new ArrayList<Edge>();

  // add all left/right edges
  for (int[] building : buildings) {
    Edge startEdge = new Edge(building[0], building[2], true);
    edges.add(startEdge);
    Edge endEdge = new Edge(building[1], building[2], false);
    edges.add(endEdge);
  }
  Collections.sort(edges, new Comparator<Edge>() {
    public int compare(Edge a, Edge b) {
      if (a.x != b.x)
        return Integer.compare(a.x, b.x);

      if (a.isStart && b.isStart) {
        return Integer.compare(b.height, a.height);
      }

      if (!a.isStart && !b.isStart) {
        return Integer.compare(a.height, b.height);
      }

      return a.isStart ? -1 : 1;
    }
  });

  // process edges
  PriorityQueue<Integer> heightHeap = new PriorityQueue<Integer>(10,
      Collections.reverseOrder());

  for (Edge edge : edges) {
    if (edge.isStart) {
      if (heightHeap.isEmpty() || edge.height > heightHeap.peek()) {
        result.add(new int[] { edge.x, edge.height });
      }
      heightHeap.add(edge.height);
    } else {
      heightHeap.remove(edge.height);

      if(heightHeap.isEmpty()){
        result.add(new int[] {edge.x, 0});
      }else if(edge.height > heightHeap.peek()){
        result.add(new int[]{edge.x, heightHeap.peek()});
      }
    }
  }

  return result;
}
```

163 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 `#`.

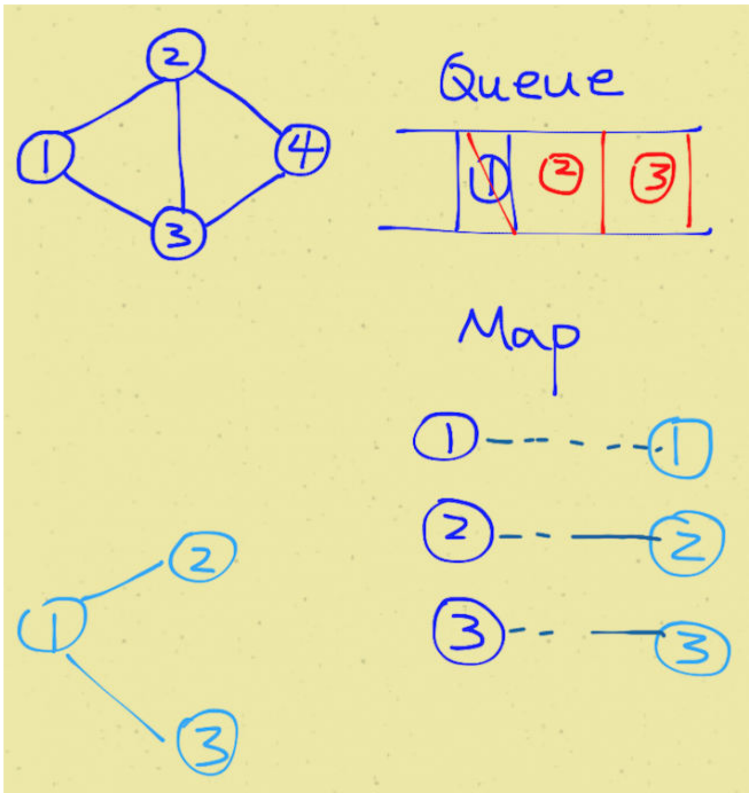
1. First node is labeled as `0`. Connect node `0` to both nodes `1` and `2`.
2. Second node is labeled as `1`. Connect node `1` to node `2`.
3. Third node is labeled as `2`. Connect node `2` to node `2` (itself), thus forming a self-cycle.

Visually, the graph looks like the following:

```
       1
      / \
     /   \
    0 --- 2
         / \
         \_/
```

163.1 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.



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

164 Course Schedule

There are a total of n courses you have to take, labeled from 0 to n - 1. Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: [0,1]. Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

For example, given 2 and [[1,0]], there are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

For another example, given 2 and [[1,0],[0,1]], there are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

164.1 Analysis

This problem can be converted to finding if a graph contains a cycle.

164.2 Java Solution 1 - BFS

This solution uses breath-first search and it is easy to understand.

```java
public boolean canFinish(int numCourses, int[][] prerequisites) {
   if(prerequisites == null){
      throw new IllegalArgumentException("illegal prerequisites array");
   }

   int len = prerequisites.length;

   if(numCourses == 0 || len == 0){
      return true;
   }

   // counter for number of prerequisites
   int[] pCounter = new int[numCourses];
   for(int i=0; i<len; i++){
      pCounter[prerequisites[i][0]]++;
   }

   //store courses that have no prerequisites
   LinkedList<Integer> queue = new LinkedList<Integer>();
   for(int i=0; i<numCourses; i++){
      if(pCounter[i]==0){
      }
   }

   // number of courses that have no prerequisites
   int numNoPre = queue.size();

   while(!queue.isEmpty()){
      int top = queue.remove();
      for(int i=0; i<len; i++){
         // if a course's prerequisite can be satisfied by a course in queue
         if(prerequisites[i][1]==top){
            pCounter[prerequisites[i][0]]--;
            if(pCounter[prerequisites[i][0]]==0){
               numNoPre++;
               queue.add(prerequisites[i][0]);
            }
         }
      }
   }

   return numNoPre == numCourses;
}
```

164.3 Java Solution 2 - DFS

```
public boolean canFinish(int numCourses, int[][] prerequisites) {
   if(prerequisites == null){
      throw new IllegalArgumentException("illegal prerequisites array");
   }

   int len = prerequisites.length;

   if(numCourses == 0 || len == 0){
      return true;
   }

   //track visited courses
   int[] visit = new int[numCourses];

   // use the map to store what courses depend on a course
   HashMap<Integer,ArrayList<Integer>> map = new
       HashMap<Integer,ArrayList<Integer>>();
   for(int[] a: prerequisites){
      if(map.containsKey(a[1])){
         map.get(a[1]).add(a[0]);
      }else{
         l.add(a[0]);
         map.put(a[1], l);
      }
   }

   for(int i=0; i<numCourses; i++){
      if(!canFinishDFS(map, visit, i))
         return false;
   }

   return true;
}

private boolean canFinishDFS(HashMap<Integer,ArrayList<Integer>> map, int[]
    visit, int i){
   if(visit[i]==-1)
      return false;
   if(visit[i]==1)
      return true;

   visit[i]=-1;
   if(map.containsKey(i)){
      for(int j: map.get(i)){
         if(!canFinishDFS(map, visit, j))
            return false;
      }
   }

   visit[i]=1;

   return true;
}
```

Topological Sort Video from Coursera.

165 Course Schedule II

This is an extension of Course Schedule. This time a valid sequence of courses is required as output.

165.1 Analysis

If we use the DFS solution of Course Schedule, a valid sequence can easily be recorded.

165.2 Java Solution

```java
public int[] findOrder(int numCourses, int[][] prerequisites) {
   if(prerequisites == null){
      throw new IllegalArgumentException("illegal prerequisites array");
   }

   int len = prerequisites.length;

   //if there is no prerequisites, return a sequence of courses
   if(len == 0){
      int[] res = new int[numCourses];
      for(int m=0; m<numCourses; m++){
         res[m]=m;
      }
      return res;
   }

   //records the number of prerequisites each course (0,...,numCourses-1)
       requires
   int[] pCounter = new int[numCourses];
   for(int i=0; i<len; i++){
      pCounter[prerequisites[i][0]]++;
   }

   //stores courses that have no prerequisites
   LinkedList<Integer> queue = new LinkedList<Integer>();
   for(int i=0; i<numCourses; i++){
      if(pCounter[i]==0){
         queue.add(i);
      }
   }
   int numNoPre = queue.size();

   //initialize result
   int[] result = new int[numCourses];
   int j=0;

   while(!queue.isEmpty()){
      int c = queue.remove();
      result[j++]=c;

      for(int i=0; i<len; i++){
         if(prerequisites[i][1]==c){
            pCounter[prerequisites[i][0]]--;
            if(pCounter[prerequisites[i][0]]==0){
               queue.add(prerequisites[i][0]);
               numNoPre++;
            }
         }

      }
   }

   //return result
   if(numNoPre==numCourses){
      return result;
   }else{
      return new int[0];
   }
}
```

166 Reconstruct Itinerary

Given a list of airline tickets represented by pairs of departure and arrival airports [from, to], reconstruct the itinerary in order. All of the tickets belong to a man who departs from JFK. Thus, the itinerary must begin with JFK.

166.1 Analysis

This is an application of Hierholzer's algorithm to find a Eulerian path.

PriorityQueue should be used instead of TreeSet, because there are duplicate entries.

166.2 Java Solution

```java
public class Solution{
  HashMap<String, PriorityQueue<String>> map = new HashMap<String,
      PriorityQueue<String>>();
  LinkedList<String> result = new LinkedList<String>();

  public List<String> findItinerary(String[][] tickets) {
    for (String[] ticket : tickets) {
      if (!map.containsKey(ticket[0])) {
        PriorityQueue<String> q = new PriorityQueue<String>();
        map.put(ticket[0], q);
      }
      map.get(ticket[0]).offer(ticket[1]);
    }

    dfs("JFK");
    return result;
  }

  public void dfs(String s) {
    PriorityQueue<String> q = map.get(s);

    while (q != null && !q.isEmpty()) {
      dfs(q.poll());
    }

    result.addFirst(s);
  }
```

167 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.

167.1 Using sort() Method

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

```java
// 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.

```java
// 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.

167.2 Using Sorted Data Structures

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

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

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).

167.3 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;
    }
```

168 Solution Merge Sort LinkedList in Java

LeetCode - Sort List:

Sort a linked list in O(n log n) time using constant space complexity.

168.1 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;
```

```java
        } 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
```

169 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

Here is a very good animation of quicksort.

```
public class QuickSort {
  public static void main(String[] args) {
    int[] x = { 9, 2, 4, 7, 3, 7, 10 };
    System.out.println(Arrays.toString(x));

    int low = 0;
    int high = x.length - 1;

    quickSort(x, low, high);
    System.out.println(Arrays.toString(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);
  }
}
```

Output:

```
9 2 4 7 3 7 10 2 3 4 7 7 9 10
```

170 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.

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 **7** 4 9 5 2 6 1

3 **4** 7 9 5 2 6 1

3 4 7 **9** 5 2 6 1

3 4 **5** 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 **6** 7 9 1

1 2 3 4 5 6 7 9

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 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
```

171 Maximum Gap

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.

171.1 Analysis

We can use a bucket-sort like algorithm to solve this problem in time of O(n) and space O(n). 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.

171.2 Java Solution

```java
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++){
      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;
}
```

172 First Missing Positive

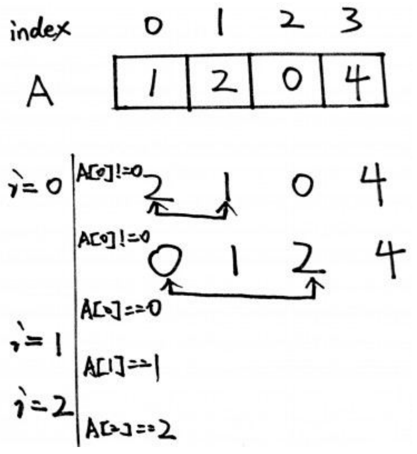
Given an unsorted integer array, find the first missing positive integer. For example, given [1,2,0] return 3 and [3,4,-1,1] return 2.

Your algorithm should run in O(n) time and uses constant space.

172.1 Analysis

This problem can solve by using a bucket-sort like algorithm. Let's consider finding first missing positive and 0 first. The key fact is that the ith element should be i, so we have: i==A[i] A[i]==A[A[i]]

For example, given an array 1,2,0,4, the algorithm does the following:



```
int firstMissingPositiveAnd0(int A[]) {
  int n = A.length;
  for (int i = 0; i < n; i++) {
    // when the ith element is not i
    while (A[i] != i) {
      // no need to swap when ith element is out of range [0,n]
      if (A[i] < 0 || A[i] >= n)
        break;

      //handle duplicate elements
      if(A[i]==A[A[i]])
                break;
      // swap elements
      A[i] = A[temp];
      A[temp] = temp;
    }
  }

  for (int i = 0; i < n; i++) {
    if (A[i] != i)
      return i;
  }

  return n;
}
```

172.2 Java Solution

This problem only considers positive numbers, so we need to shift 1 offset. The ith element is i+1.

```
public int firstMissingPositive(int[] A) {
      int n = A.length;

    for (int i = 0; i < n; i++) {
      while (A[i] != i + 1) {
        if (A[i] <= 0 || A[i] >= n)
          break;

            if(A[i]==A[A[i]-1])
                break;

        int temp = A[i];
        A[i] = A[temp - 1];
        A[temp - 1] = temp;
      }
    }

    for (int i = 0; i < n; i++){
      if (A[i] != i + 1){
        return i + 1;
      }
    }

    return n + 1;
}
```

173 Sort Colors

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

173.1 Java Solution 1 - Counting Sort

Check out this animation to understand how counting sort works.

```java
public void sortColors(int[] nums) {
   if(nums==null||nums.length<2){
      return;
   }

   int[] countArray = new int[3];
   for(int i=0; i<nums.length; i++){
      countArray[nums[i]]++;
   }

   for(int i=1; i<=2; i++){
      countArray[i]=countArray[i-1]+countArray[i];
   }

   int[] sorted = new int[nums.length];
   for(int i=0;i<nums.length; i++){
      int index = countArray[nums[i]]-1;
      countArray[nums[i]] = countArray[nums[i]]-1;
      sorted[index]=nums[i];
   }

   System.arraycopy(sorted, 0, nums, 0, nums.length);
}
```

173.2 Java Solution 2 - Improved Counting Sort

In solution 1, two arrays are created. One is for counting, and the other is for storing the sorted array (space is O(n)). We can improve the solution so that it only uses constant space. Since we already get the count of each element, we can directly project

```
public void sortColors(int[] nums) {
   if(nums==null||nums.length<2){
      return;
   }

   int[] countArray = new int[3];
   for(int i=0; i<nums.length; i++){
      countArray[nums[i]]++;
   }

   int j = 0;
   int k = 0;
   while(j<=2){
      if(countArray[j]!=0){
         nums[k++]=j;
         countArray[j] = countArray[j]-1;
      }else{
         j++;
      }
   }
}
```

174 Edit Distance in Java

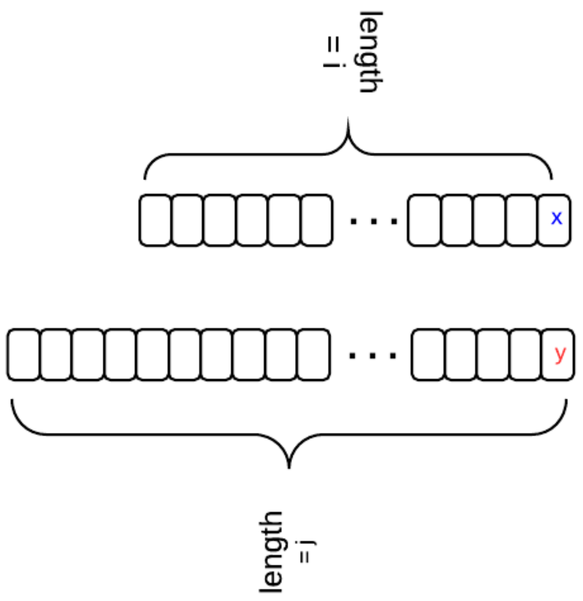
From Wiki:

> *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.

174.1 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 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

174.2 Java Code

After the analysis above, the code is just a representation of it.

```java
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];
```

175 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.

175.1 Analysis

The problem itself is very difficult to understand. It can be stated like this: Give a sequence S and T, how many distinct sub sequences from S equals to T? How do you define "distinct" subsequence? Clearly, the 'distinct' here mean different operation combination, not the final string of subsequence. Otherwise, the result is always 0 or 1. – from Jason's comment

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

175.2 Java Solution 1

Let W(i, j) stand for the number of subsequences of S(0, i) equals to 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).

```java
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()];
}
```

175.3 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()];
}
```

176 Longest Palindromic Substring

Finding the longest palindromic substring is a classic problem of coding interview. This post summarizes 3 different solutions for this problem.

176.1 Naive Approach

Naively, we can simply examine every substring and check if it is palindromic. The time complexity is O(n3̂). 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;
}
```

176.2 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 a 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(n2̂) Space O(n2̂)

```
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 a string, we can use the printTable() method to examine the table after execution. For example, if the 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 clearly see that the longest string is in cell table[1][5].

176.3 A Simple Algorithm

Time O(n2̂), 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);
}
```

176.4 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.

177 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".

177.1 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 is O(n2̂) and exceeds the time limit.

177.2 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".

177.3 Regular Expression

The problem is equivalent to matching the regular expression (leet|code)*, which

(Thanks to hdante.) Leetcode online judge does not allow using the 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");
  }
}
```

177.4 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.

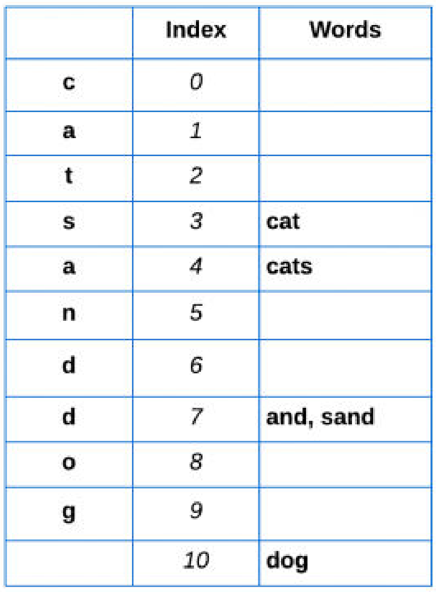
178 Word Break II

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"].

178.1 Java Solution - Dynamic Programming

This problem is very similar to Word Break. Instead of using a boolean array to track the matched positions, we need to track the actual matched 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>();
```

```
        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);
     }
  }
```

This problem is also useful for solving real problems. Assuming you want to analyze the domain names of the top 10k websites. We can use this solution to break the main

179 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.

179.1 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.

```java
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;
  }
}
```

179.2 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."

```java
public class Solution {
  public int maxSubArray(int[] A) {
    int max = A[0];
    int[] sum = new int[A.length];
    sum[0] = A[0];
```

```
      sum[i] = Math.max(A[i], sum[i - 1] + A[i]);
      max = Math.max(max, sum[i]);
    }

    return max;
  }
}
```

179.3 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.

180 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.

180.1 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(n3̂).

180.2 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 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).

181 Palindrome Partitioning

181.1 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"]
 ]
```

181.2 Depth-first Search

```
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>();//track each possible
      partition
  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);
```

```
      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;
}
```

181.3 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++) {
      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;
}
```

182 Palindrome Partitioning II

Given a string s, partition s such that every substring of the partition is a palindrome. Return the minimum cuts needed for a palindrome partitioning of s. For example, given s = "aab", return 1 since the palindrome partitioning ["aa","b"] could be produced using 1 cut.

182.1 Analysis

This problem is similar to Palindrome Partitioning. It can be efficiently solved by using dynamic programming. Unlike "Palindrome Partitioning", we need to maintain two cache arrays, one tracks the partition position and one tracks the number of minimum cut.

182.2 Java Solution

```
public int minCut(String s) {
   int n = s.length();

  boolean dp[][] = new boolean[n][n];
  int cut[] = new int[n];

  for (int j = 0; j < n; j++) {
    cut[j] = j; //set maximum # of cut
    for (int i = 0; i <= j; i++) {
      if (s.charAt(i) == s.charAt(j) && (j - i <= 1 || dp[i+1][j-1])) {
        dp[i][j] = true;

        // if need to cut, add 1 to the previous cut[i-1]
        if (i > 0){
          cut[j] = Math.min(cut[j], cut[i-1] + 1);
        }else{
        // if [0...j] is palindrome, no need to cut
          cut[j] = 0;
        }
      }
    }
  }

  return cut[n-1];
```

183 House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

183.1 Java Solution 1 - Dynamic Programming

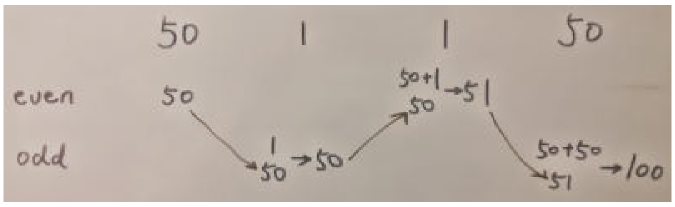
The key is to find the relation dp[i] = Math.max(dp[i-1], dp[i-2]+num[i-1]).

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

   int n = num.length;

   int[] dp = new int[n+1];
   dp[0]=0;
   dp[1]=num[0];

   for (int i=2; i<n+1; i++){
     dp[i] = Math.max(dp[i-1], dp[i-2]+num[i-1]);
   }

   return dp[n];
}
```

183.2 Java Solution 2

We can use two variables, even and odd, to track the maximum value so far as iterating the array. You can use the following example to walk through the code.

```
50 1 1 50
```



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

  int even = 0;
  int odd = 0;

  for (int i = 0; i < num.length; i++) {
    if (i % 2 == 0) {
      even += num[i];
      even = even > odd ? even : odd;
    } else {
      odd += num[i];
      odd = even > odd ? even : odd;
    }
  }

  return even > odd ? even : odd;
}
```

184 House Robber II

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

184.1 Analysis

This is an extension of House Robber. There are two cases here 1) 1st element is included and last is not included 2) 1st is not included and last is included. Therefore, we can use the similar dynamic programming approach to scan the array twice and get the larger value.

184.2 Java Solution

```java
public int rob(int[] nums) {
   if(nums==null||nums.length==0)
      return 0;

   int n = nums.length;

   if(n==1){
      return nums[0];
   }
   if(n==2){
      return Math.max(nums[1], nums[0]);
   }

   //include 1st element, and not last element
   int[] dp = new int[n+1];
   dp[0]=0;
   dp[1]=nums[0];

   for(int i=2; i<n; i++){
    }

    //not include frist element, and include last element
    int[] dr = new int[n+1];
    dr[0]=0;
    dr[1]=nums[1];

    for(int i=2; i<n; i++){
      dr[i] = Math.max(dr[i-1], dr[i-2]+nums[i]);
    }

    return Math.max(dp[n-1], dr[n-1]);
}
```

185 House Robber III

The houses form a binary tree. If the root is robbed, its left and right can not be robbed.

185.1 Analysis

Traverse down the tree recursively. We can use an array to keep 2 values: the maximum money when a root is selected and the maximum value when a root if NOT selected.

185.2 Java Solution

```java
public int rob(TreeNode root) {
   if(root == null)
      return 0;

   int[] result = helper(root);
   return Math.max(result[0], result[1]);
}

public int[] helper(TreeNode root){
   if(root == null){
      int[] result = {0, 0};
      return result;
   }

   int[] result = new int[2];
   int[] left = helper(root.left);
   int[] right = helper (root.right);

   // result[0] is when root is selected, result[1] is when not.
   result[0] = root.val + left[1] + right[1];
   result[1] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);

   return result;
}
```

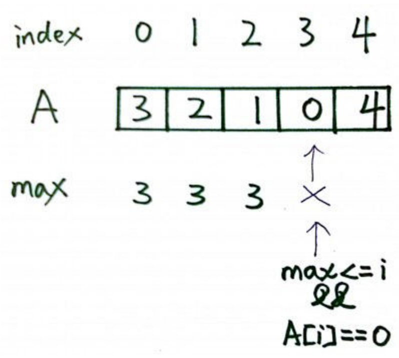
186 Jump Game

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.

186.1 Analysis

We can track the maximum index that can be reached. The key to solve this problem is to find: 1) when the current position can not reach next position (return false) , and 2) when the maximum index can reach the end (return true).

The largest index that can be reached is: i + A[i].



186.2 Java Solution

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

   int max = A[0]; //max stands for the largest index that can be reached.

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

187 Jump Game II

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.

Your goal is to reach the last index in the minimum number of jumps.

For example, given array A = [2,3,1,1,4], the minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

187.1 Analysis

This is an extension of Jump Game.

The solution is similar, but we also track the maximum steps of last jump.

187.2 Java Solution

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

  int lastReach = 0;
  int reach = 0;
  int step = 0;

  for (int i = 0; i <= reach && i < nums.length; i++) {
    //when last jump can not read current i, increase the step by 1
    if (i > lastReach) {
      step++;
      lastReach = reach;
    }
    //update the maximal jump
    reach = Math.max(reach, nums[i] + i);
  }

  if (reach < nums.length - 1)
    return 0;

  return step;
}
```

188 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.

188.1 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;
}
```

188.2 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;
}
```

189 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).

189.1 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.

189.2 Java Solution

```java
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;
}
```

190 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).

190.1 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

190.2 Java Solution

```java
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);
  }
  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;
}
```

191 Best Time to Buy and Sell Stock IV

191.1 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).

191.2 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.

191.3 Java Solution - 2D Dynamic Programming

```java
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++) {
    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];
}
```

191.4 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];
}
```

192 Dungeon Game

Example:

```
-2 (K) -3 3
-5 -10 1
10 30 -5 (P)
```

192.1 Java Solution

This problem can be solved by using dynamic programming. We maintain a 2-D table. h[i][j] is the minimum health value before he enters (i,j). h[0][0] is the value of the answer. The left part is filling in numbers to the table.

```java
public int calculateMinimumHP(int[][] dungeon) {
  int m = dungeon.length;
  int n = dungeon[0].length;

  //init dp table
  int[][] h = new int[m][n];

  h[m - 1][n - 1] = Math.max(1 - dungeon[m - 1][n - 1], 1);

  //init last row
  for (int i = m - 2; i >= 0; i--) {
    h[i][n - 1] = Math.max(h[i + 1][n - 1] - dungeon[i][n - 1], 1);
  }

  //init last column
  for (int j = n - 2; j >= 0; j--) {
    h[m - 1][j] = Math.max(h[m - 1][j + 1] - dungeon[m - 1][j], 1);
  }

  //calculate dp table
  for (int i = m - 2; i >= 0; i--) {
    for (int j = n - 2; j >= 0; j--) {
      int down = Math.max(h[i + 1][j] - dungeon[i][j], 1);
      int right = Math.max(h[i][j + 1] - dungeon[i][j], 1);
      h[i][j] = Math.min(right, down);
    }
  }
```

193 Decode Ways

A message containing letters from A-Z is being encoded to numbers using the following mapping:

'A' ->1 'B' ->2 ... 'Z' ->26

Given an encoded message containing digits, determine the total number of ways to decode it.

193.1 Analysis

193.2 Java Solution

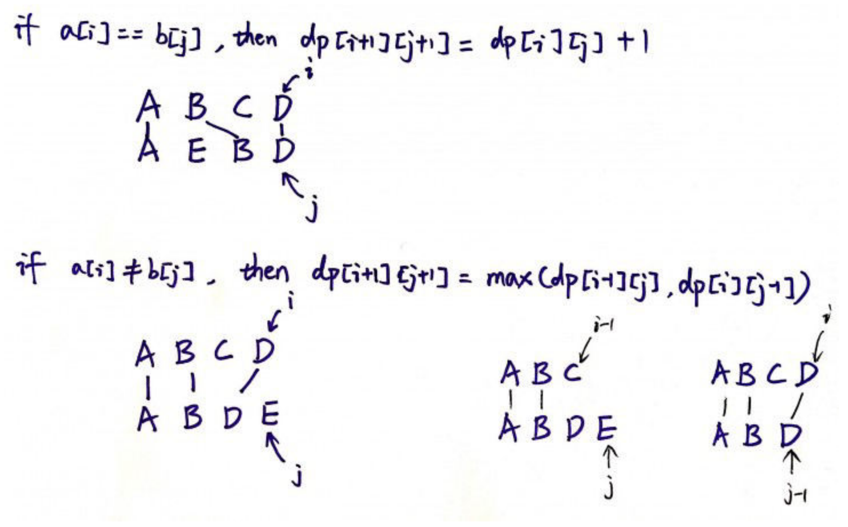
```
public int numDecodings(String s) {
   if(s==null||s.length()==0||s.equals("0"))
      return 0;


   int[] t = new int[s.length()+1];
   t[0] = 1;

   //if(s.charAt(0)!='0')
   if(isValid(s.substring(0,1)))
      t[1]=1;
   else
      t[1]=0;

   for(int i=2; i<=s.length(); i++){
      if(isValid(s.substring(i-1,i))){
         t[i]+=t[i-1];
      }

      if(isValid(s.substring(i-2,i))){
         t[i]+=t[i-2];
      }
   }

   return t[s.length()];
}

public boolean isValid(String s){
   if(s.charAt(0)=='0')
```

```
    int value = Integer.parseInt(s);
    return value>=1&&value<=26;
}
```

194 Longest Common Subsequence

The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences).

194.1 Analysis



if a[i] == b[j], then dp[i+1][j+1] = dp[i][j] + 1
if a[i] ≠ b[j], then dp[i+1][j+1] = max(dp[i+1][j], dp[i][j+1])

194.2 Java Solution

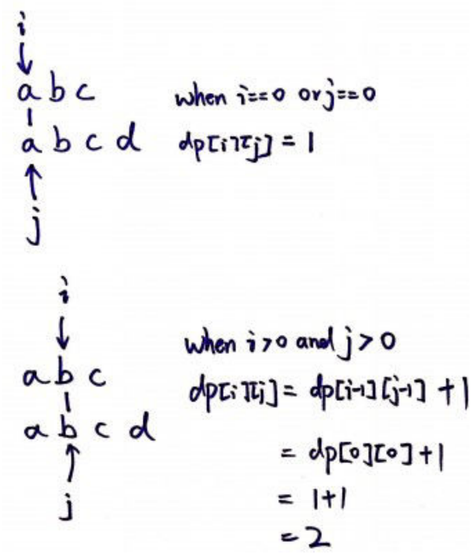
```
public static int getLongestCommonSubsequence(String a, String b){
  int m = a.length();
  int n = b.length();
  int[][] dp = new int[m+1][n+1];

  for(int i=0; i<=m; i++){
    for(int j=0; j<=n; j++){
      if(i==0 || j==0){
        dp[i][j]=0;
      }else if(a.charAt(i-1)==b.charAt(j-1)){
        dp[i][j] = 1 + dp[i-1][j-1];
      }else{
        dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
      }
  }

  return dp[m][n];
}
```

195 Longest Common Substring

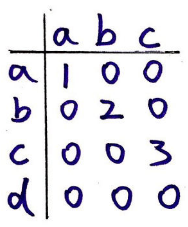
In computer science, the longest common substring problem is to find the longest string that is a substring of two or more strings.

195.1 Analysis

Given two strings a and b, let dp[i][j] be the length of the common substring ending at a[i] and b[j].



The dp table looks like the following given a="abc" and b="abcd".



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

195.2 Java Solution

```
public static int getLongestCommonSubstring(String a, String b){
  int m = a.length();
  int n = b.length();

  int max = 0;

  int[][] dp = new int[m][n];

  for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
      if(a.charAt(i) == b.charAt(j)){
        if(i==0 || j==0){
          dp[i][j]=1;
        }else{
          dp[i][j] = dp[i-1][j-1]+1;
        }

        if(max < dp[i][j])
          max = dp[i][j];
      }

    }
  }

  return max;
}
```

This is a similar problem like longest common subsequence. The difference of the solution is that for this problem when a[i]!=b[j], dp[i][j] are all zeros by default. However, in the longest common subsequence problem, dp[i][j] values are carried from the previous values, i.e., dp[i-1][j] and dp[i][j-1].

196 Longest Increasing Subsequence

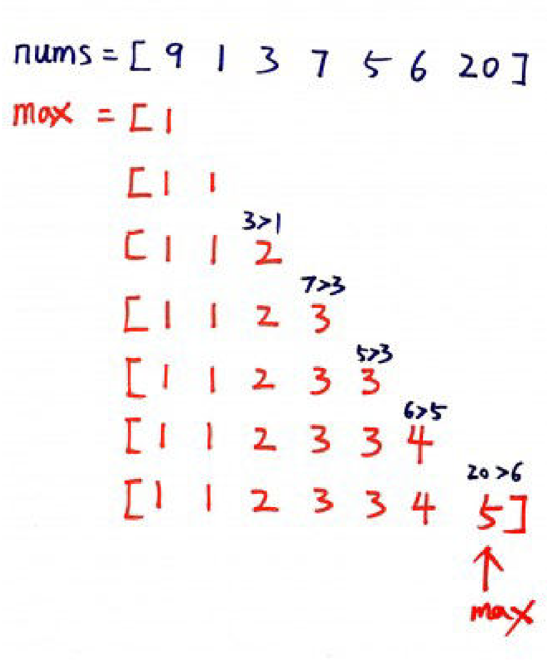
Given an unsorted array of integers, find the length of longest increasing subsequence.

For example, given [10, 9, 2, 5, 3, 7, 101, 18], the longest increasing subsequence is [2, 3, 7, 101]. Therefore the length is 4.

196.1 Java Solution 1 - Dynamic Programming

Let max[i] represent the length of the longest increasing subsequence so far. If any element before i is smaller than nums[i], then max[i] = max(max[i], max[j]+1).

Here is an example:

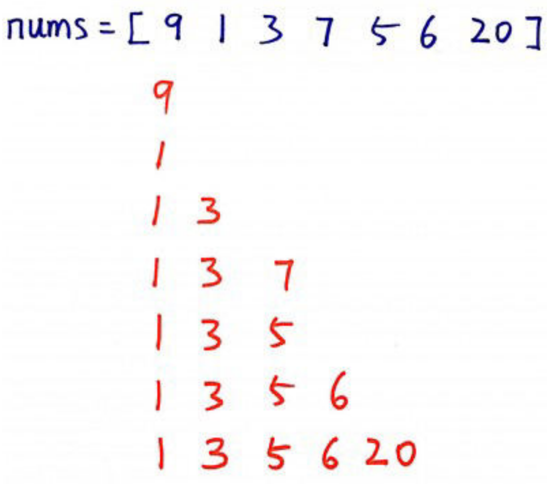


```
public int lengthOfLIS(int[] nums) {
   if(nums==null || nums.length==0)
      return 0;

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

   for(int i=0; i<nums.length; i++){
      for(int j=0; j<i;j++){
         if(nums[i]>nums[j]){
            max[i]=Math.max(max[i], max[j]+1);
         }
      }
   }

   int result = 0;
   for(int i=0; i<max.length; i++){
      if(max[i]>result)
         result = max[i];
   }
   return result;
}
```

196.2 Java Solution 2 - O(nlog(n))

We can put the increasing sequence in a list.

```
for each num in nums
    if(list.size()==0)
        add num to list
    else if(num > last element in list)
        add num to list
    else
        replace the element in the list which is the smallest but bigger than
            num
```



```
public int lengthOfLIS(int[] nums) {
      return 0;

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

   for(int num: nums){
      if(list.size()==0){
         list.add(num);
      }else if(num>list.get(list.size()-1)){
         list.add(num);
      }else{
         int i=0;
         int j=list.size()-1;

         while(i<j){
            int mid = (i+j)/2;
            if(list.get(mid) < num){
               i=mid+1;
            }else{
               j=mid;
            }
         }

         list.set(j, num);
      }
   }

   return list.size();
}
```

197 Coin Change

You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

197.1 Java Solution 1 - Dynamic Programming

```
Let dp[v] to be the minimum number of coins required to get the amount v.
dp[i+a_coin] = min(dp[i+a_coin], dp[i]+1) if dp[i] is reachable.
dp[i+a_coin] = dp[i+a_coin] is dp[i] is not reachable.
We initially set dp[i] to be MAX_VALUE.
```

Here is the Java code:

```java
public int coinChange(int[] coins, int amount) {
   if(amount==0) return 0;

   int[] dp = new int [amount+1];
   dp[0]=0; // do not need any coin to get 0 amount
   for(int i=1;i<=amount; i++)
      dp[i]= Integer.MAX_VALUE;

   for(int i=0; i<=amount; i++){
      for(int coin: coins){
         if(i+coin <=amount){
            if(dp[i]==Integer.MAX_VALUE){
               dp[i+coin] = dp[i+coin];
            }else{
               dp[i+coin] = Math.min(dp[i+coin], dp[i]+1);
            }
         }
      }
   }

   if(dp[amount] >= Integer.MAX_VALUE)
      return -1;

   return dp[amount];
}
```

197.2 Java Solution 2 - Breath First Search (BFS)

Most dynamic programming problems can be solved by using BFS.

We can view this problem as going to a target position with steps that are allows in the array coins. We maintain two queues: one of the amount so far and the other for the minimal steps. The time is too much because of the contains method take n and total time is O(n3̂).

```
public int coinChange(int[] coins, int amount) {
  if (amount == 0)
    return 0;

  LinkedList<Integer> amountQueue = new LinkedList<Integer>();
  LinkedList<Integer> stepQueue = new LinkedList<Integer>();

  // to get 0, 0 step is required
  amountQueue.offer(0);
  stepQueue.offer(0);

  while (amountQueue.size() > 0) {
    int temp = amountQueue.poll();
    int step = stepQueue.poll();

    if (temp == amount)
      return step;

    for (int coin : coins) {
      if (temp > amount) {
        continue;
      } else {
        if (!amountQueue.contains(temp + coin)) {
          amountQueue.offer(temp + coin);
          stepQueue.offer(step + 1);
        }
      }
    }
  }

  return -1;
}
```

198 Single Number

The problem:

> *Given an array of integers, every element appears twice except for one. Find that single one.*

198.1 Java Solution 1

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
public int singleNumber(int[] A) {
  int x = 0;
  for (int a : A) {
    x = x ^ a;
  }
  return x;
}
```

198.2 Java Solution 2

```java
public int singleNumber(int[] A) {
  HashSet<Integer> set = new HashSet<Integer>();
  for (int n : A) {
    if (!set.add(n))
      set.remove(n);
  }
  Iterator<Integer> it = set.iterator();
  return it.next();
}
```

The question now is do you know any other ways to do this?

199 Single Number II

199.1 Problem

Given an array of integers, every element appears three times except for one. Find that single one.

199.2 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;
}
```

200 Twitter Codility Problem Max Binary Gap

Problem: Get maximum binary Gap.
For example, 9's binary form is 1001, the gap is 2.

200.1 Java Solution 1

An integer x & 1 will get the last digit of the integer.

```
public static int getGap(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;
}
```

Time is O(n).

200.2 Java Solution 2

```
public static int getGap(int N) {
  int pre = -1;
  int len = 0:
```

```
  while (N > 0) {
    int k = N & -N;

    int curr = (int) Math.log(k);

    N = N & (N - 1);

    if (pre != -1 && Math.abs(curr - pre) > len) {
      len = Math.abs(curr - pre) + 1;
    }
    pre = curr;
  }

  return len;
}
```

Time is O(log(n)).

201 Number of 1 Bits

201.1 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.

201.2 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;
}
```

202 Reverse Bits

202.1 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

202.2 Java Solution

```java
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;
}
```

203 Repeated DNA Sequences

203.1 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"].

203.2 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.

```java
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));
      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;
}
```

204 Bitwise AND of Numbers Range

204.1 Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive. For example, given the range [5, 7], you should return 4. Java Solution

The key to solve this problem is bitwise AND consecutive numbers. You can use the following example to walk through the code.

```
8 4 2 1
---------------
5 | 0 1 0 1
6 | 0 1 1 0
7 | 0 1 1 1
```

```java
public int rangeBitwiseAnd(int m, int n) {
    while (n > m) {
        n = n & n - 1;
    }
    return m & n;
}
```

205 Power of Two

Given an integer, write a function to determine if it is a power of two.

205.1 Analysis

If a number is power of 2, it's binary form should be 10...0. So if we right shift a bit of the number and then left shift a bit, the value should be the same when the number >= 10(i.e.,2).

205.2 Java Solution

```java
public boolean isPowerOfTwo(int n) {
   if(n<=0)
      return false;

   while(n>2){
      int t = n>>1;
      int c = t<<1;

      if(n-c != 0)
         return false;

      n = n>>1;
   }

   return true;
}
```

206 Counting Bits

Given a non negative integer number num. For every numbers i in the range $0 \leq i \leq$ num calculate the number of 1's in their binary representation and return them as an array.

Example:

For num = 5 you should return [0,1,1,2,1,2].

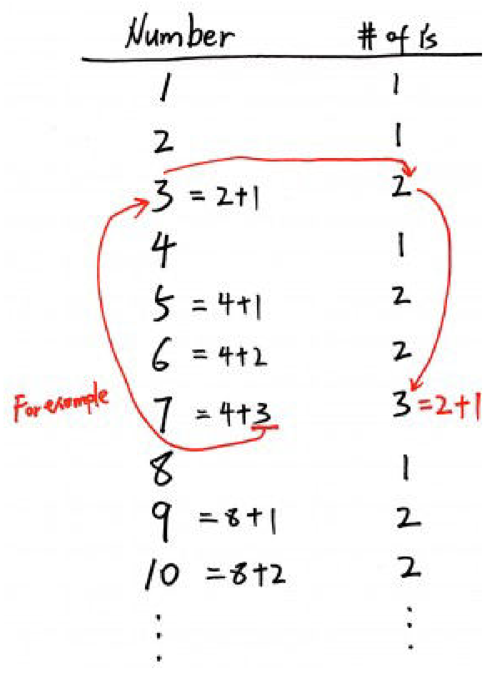
206.1 Naive Solution

We can simply count bits for each number like the following:

```
public int[] countBits(int num) {
   int[] result = new int[num+1];

   for(int i=0; i<=num; i++){
      result[i] = countEach(i);
   }

   return result;
}

public int countEach(int num){
   int result = 0;

   while(num!=0){
      if(num%2==1){
         result++;
      }
      num = num/2;
   }

   return result;
}
```

206.2 Improved Solution

For number 2(10), 4(100), 8(1000), 16(10000), ..., the number of 1's is 1. Any other number can be converted to be $2\hat{m} + x$. For example, 9=8+1, 10=8+2. The number of



```java
public int[] countBits(int num) {
   int[] result = new int[num+1];

   int p = 1; //p tracks the index for number x
   int pow = 1;
   for(int i=1; i<=num; i++){
      if(i==pow){
         result[i] = 1;
         pow <<= 1;
         p = 1;
      }else{
         result[i] = result[p]+1;
         p++;
      }

   }

   return result;
}
```

207 Maximum Product of Word Lengths

Given a string array words, find the maximum value of length(word[i]) * length(word[j]) where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

207.1 Java Solution

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

   int[] arr = new int[words.length];
   for(int i=0; i<words.length; i++){
      for(int j=0; j<words[i].length(); j++){
         char c = words[i].charAt(j);
         arr[i] |= (1<< (c-'a'));
      }
   }

   int result = 0;

   for(int i=0; i<words.length; i++){
      for(int j=i+1; j<words.length; j++){
         if((arr[i] & arr[j]) == 0){
            result = Math.max(result, words[i].length()*words[j].length());
         }
      }
   }

   return result;
}
```

208 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].
```

208.1 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.

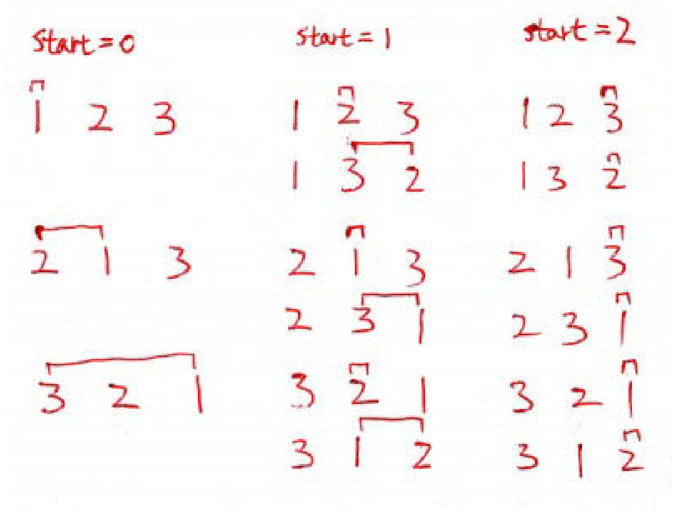
```
public ArrayList<ArrayList<Integer>> permute(int[] num) {
  ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

  //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]);
        current.add(temp);

        //System.out.println(temp);

        // - remove num[i] add
        l.remove(j);
      }
    }

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

  return result;
}
```

208.2 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;
}
```

```java
private void swap(int[] a, int i, int j) {
  int temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```



209 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].
```

209.1 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.

209.2 Java Solution 1

```java
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>();
```

```
    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;
}
```

209.3 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!

210 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.)

210.1 Java Solution 1

```java
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
      // update k
      k = k % mod;

      // get number according to curIndex
      result += numberList.get(curIndex);
      // remove from list
      numberList.remove(curIndex);
    }

    return result.toString();
  }
}
```

210.2 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();
  }
```

211 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:

```
"((()))", "(()())", "(())()", "()(())", "()()()"
```

211.1 Java Solution 1 - DFS

This solution is simple and clear.

```
public List<String> generateParenthesis(int n) {
   ArrayList<String> result = new ArrayList<String>();
   dfs(result, "", n, n);
   return result;
}
/*
left and right represents the remaining number of ( and ) that need to be
    added.
When left > right, there are more ")" placed than "(". Such cases are wrong
    and the method stops.
*/
public void dfs(ArrayList<String> result, String s, int left, int right){
   if(left > right)
      return;

   if(left==0&&right==0){
      result.add(s);
      return;
   }

   if(left>0){
      dfs(result, s+"(", left-1, right);
   }

   if(right>0){
      dfs(result, s+")", left, right-1);
   }
}
```

211.2 Java Solution 2

This solution looks more complicated. ,You can use n=2 to walk though the code.

```
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;
}
```

212 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]
```

212.1 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.

212.2 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);
```

```
        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);
    }
}
```

213 Combination Sum II

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T. Each number in C may only be used ONCE in the combination.

Note: 1) All numbers (including target) will be positive integers. 2) Elements in a combination (a1, a2, ... , ak) must be in non-descending order. (ie, a1 $\leq$ a2 $\leq$ ... $\leq$ ak). 3) The solution set must not contain duplicate combinations.

213.1 Java Solution

This problem is an extension of Combination Sum. The difference is one number in the array can only be used ONCE.

```java
public List<ArrayList<Integer>> combinationSum2(int[] num, int target) {
   ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
   if(num == null || num.length == 0)
      return result;

   Arrays.sort(num);

   ArrayList<Integer> temp = new ArrayList<Integer>();
   getCombination(num, 0, target, temp, result);

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

   //remove duplicate lists
   result.clear();
   result.addAll(set);

   return result;
}

public void getCombination(int[] num, int start, int target,
    ArrayList<Integer> temp, ArrayList<ArrayList<Integer>> result){
   if(target == 0){
      ArrayList<Integer> t = new ArrayList<Integer>(temp);
      result.add(t);
      return;
   }

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

```
            continue;

        temp.add(num[i]);
        getCombination(num, i+1, target-num[i], temp, result);
        temp.remove(temp.size()-1);
    }
}
```

214 Combination Sum III

Find all possible combinations of k numbers that add up to a number n, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Ensure that numbers within the set are sorted in ascending order.

Example 1: Input: k = 3, n = 7 Output: [[1,2,4]] Example 2: Input: k = 3, n = 9 Output: [[1,2,6], [1,3,5], [2,3,4]]

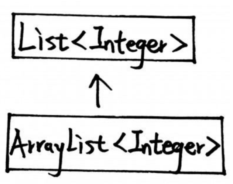
214.1 Analysis

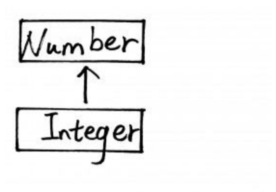
Related problems: Combination Sum, Combination Sum II.

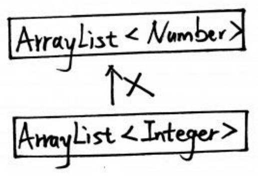
214.2 Java Solution

```java
public List<List<Integer>> combinationSum3(int k, int n) {
   List<List<Integer>> result = new ArrayList<List<Integer>>();
   List<Integer> list = new ArrayList<Integer>();
   dfs(result, 1, n, list, k);
   return result;
}

public void dfs(List<List<Integer>> result, int start, int sum, List<Integer>
    list, int k){
   if(sum==0 && list.size()==k){
      List<Integer> temp = new ArrayList<Integer>();
      temp.addAll(list);
      result.add(temp);
   }

   for(int i=start; i<=9; i++){
      if(sum-i<0) break;
      if(list.size()>k) break;

      list.add(i);
      dfs(result, i+1, sum-i, list, k);
      list.remove(list.size()-1);
   }
}
```

Integer is a subclass of Number and ArrayList is a subclass of List. But ArrayList is not a subclass of ArrayList.







215 Combinations

215.1 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],
]
```

215.2 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>();
      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);
}
```

215.3 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);
  }
}
```

216 Letter Combinations of a Phone Number

Given a digit string, return all possible letter combinations that the number could represent. (Check out your cellphone to see the mappings) Input:Digit string "23", Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

216.1 Analysis

This problem can be solves by a typical DFS algorithm. DFS problems are very similar and can be solved by using a simple recursion. Check out the index page to see other DFS problems.

216.2 Java Solution

```java
public List<String> letterCombinations(String digits) {
   HashMap<Integer, String> map = new HashMap<Integer, String>();
   map.put(2, "abc");
   map.put(3, "def");
   map.put(4, "ghi");
   map.put(5, "jkl");
   map.put(6, "mno");
   map.put(7, "pqrs");
   map.put(8, "tuv");
   map.put(9, "wxyz");
   map.put(0, "");

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

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

   ArrayList<Character> temp = new ArrayList<Character>();
   getString(digits, temp, result, map);

   return result;
}

public void getString(String digits, ArrayList<Character> temp,
          if(digits.length() == 0){
             char[] arr = new char[temp.size()];
             for(int i=0; i<temp.size(); i++){
                arr[i] = temp.get(i);
             }
             result.add(String.valueOf(arr));
             return;
          }

          Integer curr = Integer.valueOf(digits.substring(0,1));
          String letters = map.get(curr);
          for(int i=0; i<letters.length(); i++){
             temp.add(letters.charAt(i));
             getString(digits.substring(1), temp, result, map);
             temp.remove(temp.size()-1);
          }
       }
```

217 Restore IP Addresses

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example: given "25525511135",return ["255.255.11.135", "255.255.111.35"].

217.1 Java Solution

This is a typical search problem and it can be solved by using DFS.

```java
public List<String> restoreIpAddresses(String s) {
   ArrayList<ArrayList<String>> result = new ArrayList<ArrayList<String>>();
   ArrayList<String> t = new ArrayList<String>();
   dfs(result, s, 0, t);

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

   for(ArrayList<String> l: result){
      StringBuilder sb = new StringBuilder();
      for(String str: l){
         sb.append(str+".");
      }
      sb.setLength(sb.length() - 1);
      finalResult.add(sb.toString());
   }

   return finalResult;
}

public void dfs(ArrayList<ArrayList<String>> result, String s, int start,
    ArrayList<String> t){
   //if already get 4 numbers, but s is not consumed, return
   if(t.size()>=4 && start!=s.length())
      return;

   //make sure t's size + remaining string's length >=4
   if((t.size()+s.length()-start+1)<4)
      return;

   //t's size is 4 and no remaining part that is not consumed.
   if(t.size()==4 && start==s.length()){
      ArrayList<String> temp = new ArrayList<String>(t);
```

```
        return;
    }

    for(int i=1; i<=3; i++){
        //make sure the index is within the boundary
        if(start+i>s.length())
            break;

        String sub = s.substring(start, start+i);
        //handle case like 001. i.e., if length > 1 and first char is 0, ignore
             the case.
        if(i>1 && s.charAt(start)=='0'){
            break;
        }

        //make sure each number <= 255
        if(Integer.valueOf(sub)>255)
            break;

        t.add(sub);
        dfs(result, s, start+i, t);
        t.remove(t.size()-1);
    }
}
```

218 Reverse Integer

LeetCode - Reverse Integer:

> *Reverse digits of an integer. Example1: x = 123, return 321 Example2: x = -123, return -321*

218.1 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.

218.2 Efficient Approach

Actually, this can be done by using the following code.

```java
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;
}
```

218.3 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;
}
```

218.4 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.

219 Palindrome Number

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

219.1 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.

219.2 Java Solution

```java
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;
  }
}
```

220 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.

220.1 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:

```java
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).

220.2 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

```java
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));
}
```

220.3 In this solution, we can handle cases that x <0 and n <0. This solution actually takes more time than the first solution. Why? 3. 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
```

```
    result = 1 / result;

  return result;
}
```

220.4 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);
  }
}
```