Cracking Coding Interview like a Pro: A 10-Day Interview Course

Cracking Coding Interview like a Pro: A 10-Day Interview Course

Introduction

You have been sitting nervously in the interview room, waiting for the interviewer to give you a coding question. You have practiced why you want to join the company and explained your previous projects and experiences. You are deeply worried because you know that, in a matter of seconds, you're going to hit the much dreaded interview challenge: the coding question. Suddenly, you panic. You couldn't remember any data structure or algorithms you studied. Vivid memories of your last tragic interview come flooding back …

Whoa! Slow down! Let's take a deep breath and relax. You may not ever learn to love working on algorithm and coding question, but with a bit of practicing and thinking, you will be able to sail through this part of the interview just as easily as talking about your previous project. This book will tell you how.

Coding questions come in all shapes and complexities, from the simple straightforward question designed to test how familiar you are with basic data structure, to vague and complicated problems that involves multiple solutions and different trade-offs. However, they all have one thing in common: they test a candidate's aptitude for software engineering.

Who this book is for?

This book is for those who are planning to get a job in software engineering, no matter whether you are a student or an experienced professional. Multiple rounds of coding interviews are a standard way to access the engineering ability, thus the performance in these interview directly affects whether you will get an offer and if so, how good the offer package will be.

Who is the author?

I am a software engineer working in San Francisco. I have got many software engineer job offers, from big famous companies to promising

startups, including Facebook, Google, Pinterest, Stripe, Salesforce, Oracle, eBay etc. I spent my college years studying liberal arts but when I graduated, I found the chance to get a high-paying job was slim. So I enrolled at the Master program of Information Technology at Carnegie Mellon University to study computer science. At the time of graduation, I have got many job offers mostly due to my efforts and strategies to prepare for Engineering interviews.
When I was in your position, I remember not knowing software engineering was 100% right for me, but what I did understand technology is a future and software engineering is a strong career. As it turned out, software engineering was my niche and I am now privileged enough to share all my hindsight in this field.

Everything I wish someone would have told me about how to prepare coding interviews is included in the very applicable chapters of this book. All your anxieties and fear about coding interviews will melt away and you will be motivated, confident and ready to conquer the

coding questions! I hope you truly enjoy the book.

What is the most effective way of reading this book?

This book is intended to be a 10 day full time course and it is ideal to complete one chapter per day. The best way to read the book is as follows:

1) Spend 20 minutes to learn and refresh your memory about basics in the 'Basics' question. This is the foundation of solving coding problems.
2) For each coding problem, please spend 20 minutes by yourself to try to figure it out. Then move on to read the solutions and practice coding by yourself. I hope you understand the code thoroughly, because the more you understand, the less you need to memorize. All the code is written in Java because Java is very easy to understand.

Table of Content

---------- Dynamic Programming I

Day 8 ----------

---------- Dynamic Programming II

Day 9 ----------

---------- Dynamic Programming III

Day 10 ----------

---------- Time Complexity Analysis

Day 1 : Array

Array-based questions are very common in coding interviews. This chapter offers some of the very best interview questions to practice. After practicing these questions, you will feel very comfortable in handling array based questions.

Question 1:

Description:

Rotate an array by n steps. If the array is [1, 2, 3, 4, 5, 6, 7] and k = 3, the array becomes [5, 6, 7, 1, 2, 3, 4]. Please note that k is larger than the length of the array.

Algorithm:

1. If k is larger than the length of the array, we can take k%array_length.
2. Reverse the array between [0, array_length-k-1]
3. Reverse the array between [array_length -k, array_length -1]
4. Reverse the whole array

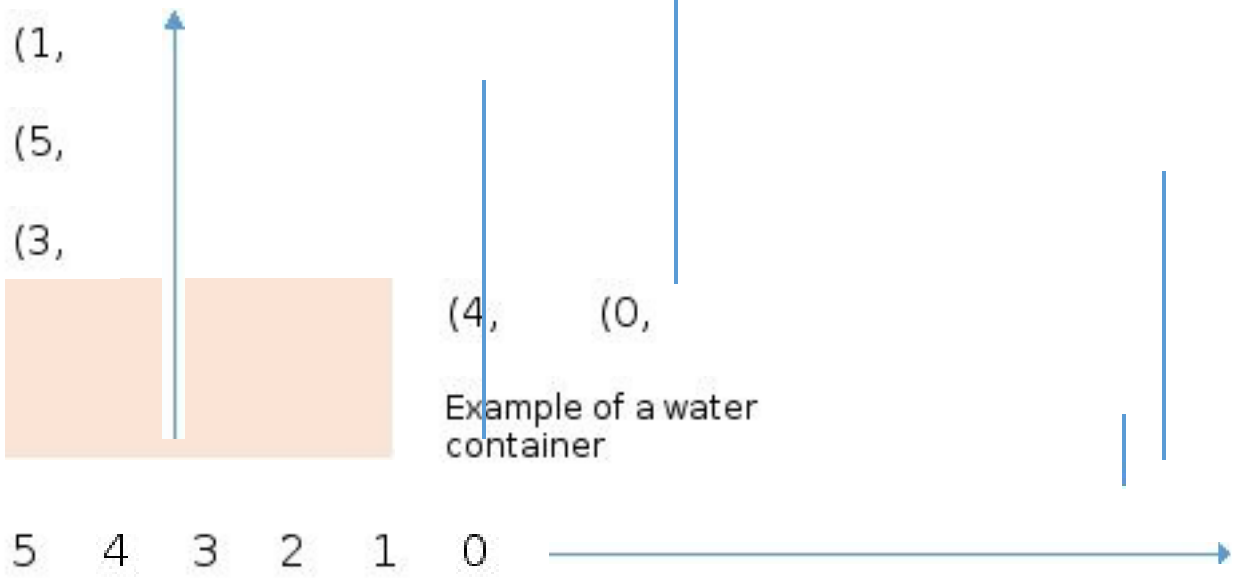
Code:

```
public void rotate(int[] nums, int k) {
    k=k%nums.length;
    if (nums==null||nums.length<=k) {
            return;
    }

    reverse(nums, 0, nums.length-k-1);
    reverse(nums, nums.length-k, nums.length-1);
    reverse(nums, 0, nums.length-1);
}
public void reverse(int[] nums, int start, int end) {
    while(start<end) {
            int t= nums[start];
                nums[start]=nums[end];
                nums[end]=t;
                start++;
                end--;
        }
}
```

Question 2:

Description:

(2, This is a very interesting coding question called ‘Container with Most Water”. Given an array of non-negative integers representing a line height, find two lines which forms a container that contains the most water.



Algorithm:

1. The area of the water container is the index gap * the lower height, because the amount of water the container could hold depends on the lower height of the two sides.

2. Keep tracking the max. Have two pointers on the left and right. If the left height is lower than the right heigher, move the left pointer because the only possibility to get a bigger area is to increase the left pointer. Otherwise, increase the right pointer.

Code:

```
public int maxArea(int[] height) {
        if (height==null||height.length==0)
            return 0;
    int i=0, j= height.length-1;
    int max = 0;
    while(i<j) {
        int area = (j-i)*Math.min(height[i], height[j]);
        max = Math.max(area, max);
        if (height[i]<height[j]) {
          i++;
        } else {
          j--;
        }
    }
    return max;
}
```

Question 3:

Description:

A sorted array is rotated. Find an element in the array. You may assume no duplicate exists in the array. For instance, find 7 in [4, 5, 6, 7, 0, 1, 2].

Algorithm:

This is a revised binary search.

1. Find the mid element, if it is the number, return the index
2. Otherwise, handle two situations: the left side is sorted or the right side is sorted. At least one side is sorted.
3. Continue with the binary search.

Code:

```
public int search(int[] nums, int target) {
    int i = 0, j= nums.length-1;
    while(i<=j) {
            int mid = (i+j)/2;
            if (nums[mid]==target) {
                    return mid;
            }
```

```
                if (nums[i]<nums[mid]) {
                        //left side sorted
                        if (target>=nums[i] &&
target<nums[mid]) {
                                j=mid-1;
                        } else {
                                i=mid+1;
                        }
                } else if (nums[mid]<nums[j]){
                        //right side sorted
                        if
(target>nums[mid]&&target<=nums[j]) {
                                i=mid+1;
                        } else {
                                j=mid-1;
                        }
                } else if (nums[i]==nums[mid]){
                        i = mid+1;
                } else if (nums[j]==nums[mid]) {
                        j=mid-1;
                }
        }
         return -1;
    }
```

Question 4:

Description:

Solving a Sudoku puzzle is a very interesting and popular interview question.
https://en.wikipedia.org/wiki/Sudoku has a clear explanation of what is Sudoku.
Write a program to solve the Sudoku puzzle by filling the empty cells.

Algorithm:

This is typical backtracking algorithm. It is basically DFS: go deep by trying to fill in a value until the board violates Sudoku rules. In that case, try the next value.

Code:

```
public void solveSudoku(char[][] board) {
        traverse(board, 0);
}
public boolean traverse(char[][] board, int count) {
    if (count==81) return true;
  int i = count/9, j = count%9;
    if (board[i][j]!='.') {
```

```
                    return traverse(board, count+1);
                }
                for (int k=1;k<=9;k++) {
                    board[i][j] = (char)('0'+k);
                    if (isValid(board)) {
                      if (traverse(board, count+1)) {
                            return true;
                      }
                    }
                }
                board[i][j] ='.';
                return false;
            }

    private boolean isValid(char[][] board) {
        HashSet<Character> set = new HashSet<Character>
();
                //row
                for(int r=0;r<9;r++) {
                   set.clear();
                        for(int c=0;c<9;c++) {
                     if (board[r][c]=='.')
                   continue;
                            if (set.contains(board[r][c])) return
false;
                                set.add(board[r][c]);
                        }
```

```
                }
                //col
                for(int c=0;c<9;c++) {
                        set.clear();
                        for(int r=0;r<9;r++) {
                           if (board[r][c]=='.') continue;
                                if
(set.contains(board[r][c])) return false;
                                        set.add(board[r][c]);
                        }
                }
                //9 grid
                for (int r= 0;r<9;r+=3) {
                        for (int c= 0;c<9;c+=3) {
                                set.clear();
                                for (int i=r;i<r+3;i++)
{
                                                for (int
j=c;j<c+3;j++) {
                                                        if (board[i]
[j]=='.') continue;
                                                                if
(set.contains(board[i][j]))
                                                        return false;

set.add(board[i][j]);
                                                }
```

```
                                    }
                                }
                            }
                            return true;
                        }
```

Question 5:

Description:

Given a matrix, print all values in spinal order.

1	2	3
4	5	6
7	8	9

Print out: 1, 2, 3, 6, 9, 8, 7, 4, 5

Algorithm:

Keep two pointers: row and column. Increase and decrease these pointers based on the direction of spinal traversal.

Code:

```
    public List<Integer> spiralOrder(int[][] arr) {
        List<Integer> res = new ArrayList<Integer>();
        if (arr==null||arr.length==0) return res;
        int rowT = 0, colL=0, rowB = arr.length-1, colR =
arr[0].length-1;
        while(true) {
            for (int i=colL;i<=colR;i++) {
                res.add(arr[rowT][i]);
```

```
            }
            rowT++;
            if (checkoutboundary(rowT, rowB, colL, colR,
arr)) break;
            for (int i=rowT;i<=rowB;i++) {
                res.add(arr[i][colR]);
            }
            colR--;
            if (checkoutboundary(rowT, rowB, colL, colR,
arr)) break;
            for (int i=colR;i>=colL;i--) {
                res.add(arr[rowB][i]);
            }
            rowB--;
            if (checkoutboundary(rowT, rowB, colL, colR,
arr)) break;
            for (int i=rowB;i>=rowT;i--) {
                res.add(arr[i][colL]);
            }
            colL++;
            if (checkoutboundary(rowT, rowB, colL, colR,
arr)) break;
        }
        return res;
    }
        private boolean checkoutboundary(int r1, int r2,
int c1, int c2,
```

```
        int[][] arr) {
    return r1>r2 ||c1>c2;
}
```

Question 6:

Description:

Given a sorted matrix (integers in each row are sorted from left to right and in each column are sorted from top to bottom), find whether this matrix contains a value.

-111	-20	-3	4
5	6	9	18
19	56	98	101

Searching for 6 returns true, 190 returns false.

Algorithm:

Start from the top right corner, if smaller than this value, decrease row value; otherwise, increase column value

Code:

```
    public boolean searchMatrix(int[][] matrix, int
target) {
      int i=0, j = matrix[0].length-1;
      while(i<matrix.length&&j>=0) {
            if (matrix[i][j]==target){
```

```
                    return true;
                } else if (matrix[i][j]>target) {
                    j--;
                } else {
                    i++;
                }
        }
        return false;
}
```

Question 7:

Description:

Find the kth largest element in an unsorted array. For example, if the array is [7, 0, -9, 5, 4] and k = 2, return 7.

Algorithm:

This algorithm is called 'Selection Rank' and the idea is very similar to 'quick sort'. It is easier to think about nth smallest than kth largest so I am using (nums.length-k+1) smallest to explain the algorithm and code.

1. Pick a random element in the array and use it as 'pivot'. Move all elements smaller than pivot to one side of the array and all elements larger to the other side.
2. If there are exactly (nums.length-k+1) element on the right, then you find the element.
3. Otherwise, if the right side is bigger than (nums.length-k+1), repeat the algorithm on right. Otherwise, repeat on the left.

Code:

```
    public int findKthLargest(int[] nums, int k) {
        if (nums==null||nums.length==0||nums.length<k)
return -1;
        return findKth(nums, nums.length-k+1, 0,
nums.length-1);
    }
    public int findKth(int[] nums, int k, int a, int b) {
                        if (a==b) return nums[a];
                        int i = a-1, j = b+1;
                        int m = nums[(a+b)/2];
                        while(true) {
                                while(nums[++i]<m);
                                while(nums[--j]>m);
                                if (i>=j) {
                                        break;
                                }
                                swap(nums, i, j);
                        }
                        if (i==j)
                                i=i+1;
                        return k<=i ? findKth(nums, k, a, j) :
findKth(nums, k, i, b);
                }
                public void swap(int[] arr, int i, int j) {
```

```
            int temp = arr[i];
            arr[i]=arr[j];
            arr[j]=temp;
}
```

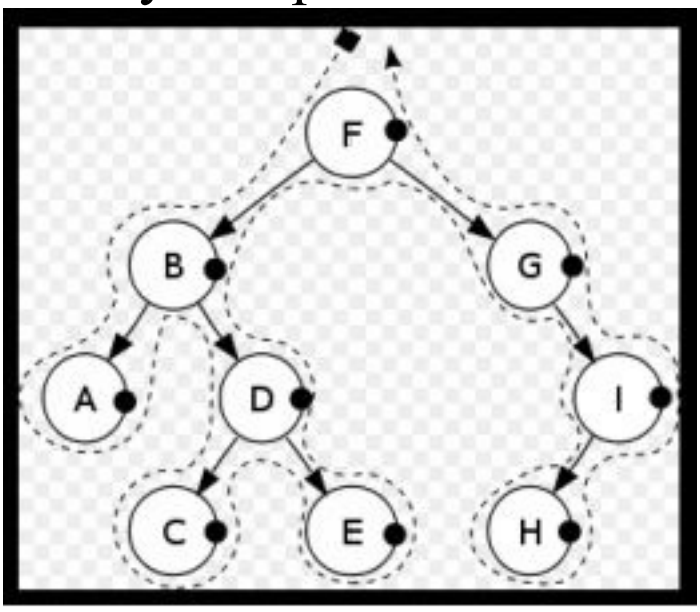
Day 2: Tree

Tree is a widely tested area in coding interviews. The key concept about traversal and recursion in trees is asked repetitively in many companies' interview questions. I found as a start, the best material is http://cslibrary.stanford.edu/110/BinaryTrees.html. You shall get yourself familiar with this material and practice all the questions at least once.

Question 1:

Description:

Binary tree post-order traversal.



The post order traversal prints A, C, E, D, B, H, I, G, F

(source of the image:

https://en.wikipedia.org/wiki/Tree_traversal#/medi

Algorithm:

During a coding interview, the interviewer is looking for an iterative solution.

The straightforward solution is to use Stack.

1. Put the root node in
2. If the peeking node has unvisited left child, put the left child in
3. If the peeking node has unvisited right child (no left child), put the right child in
4. If the peeking node has no unvisited children, pop the node

Code:

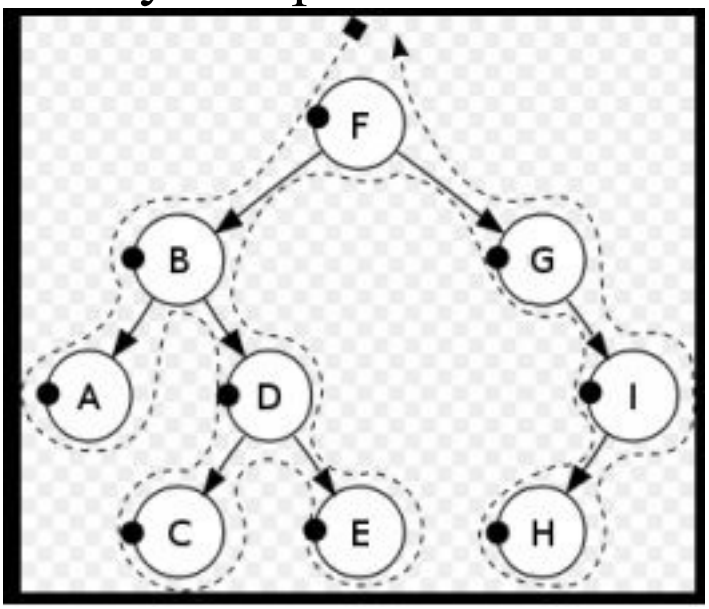
```
  public List<Integer≥ postorderTraversal(TreeNode
root) {
      ArrayList<Integer> l = new ArrayList<Integer>();
                  if (root==null) {
                        return l;
                  }
                  Stack<TreeNode> s= new
Stack<TreeNode>();
                  HashSet<TreeNode> set = new
HashSet<TreeNode>();
```

```
                    s.push(root);
                    set.add(root);
                    while(!s.isEmpty()) {
                            TreeNode current = s.peek();
                            if (current.left!=null &&
!set.contains(current.left)) {
                                    s.push(current.left);
                            } else if (current.right!=null
                        && !set.contains(current.right)) {
                                    s.push(current.right);
                            } else {
                                    TreeNode n = s.pop();
                                    set.add(n);
                                    l.add(n.val);
                            }
                    }
                    return l;
    }
```

Question 2:

Description:

Binary tree pre-order traversal.



Pre-order traversal prints: F, B, A, D, C, E, G, I, H.

(source of image: https://en.wikipedia.org/wiki/Tree_traversal#/medi

Algorithm:

Use a stack to keep the order of nodes to pop out.

1. Push in the root node
2. Pop the node
3. If the node has right child, push the right child in. If the node has left child, push the left child in.

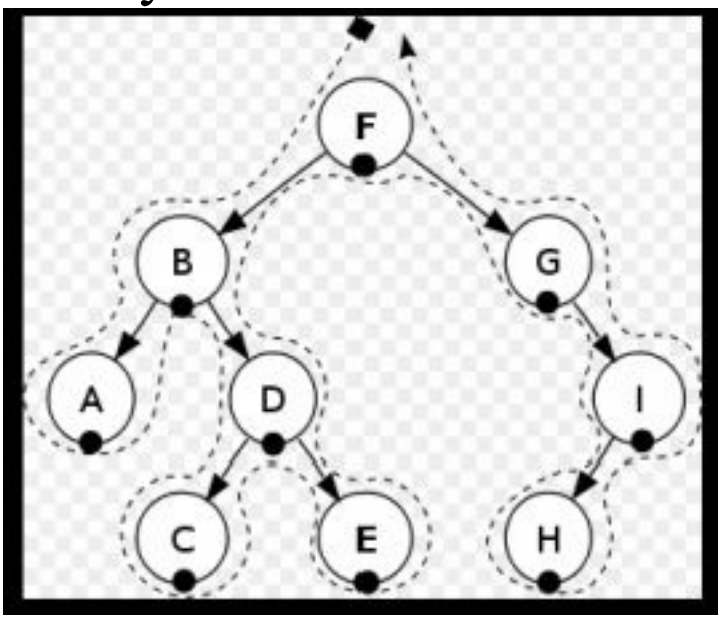
Code:

```
public List<Integer≥ preorderTraversal(TreeNode root) {
        List<Integer> l = new ArrayList<Integer>();
        traverse(root, l);
        return l;
}
public void traverse(TreeNode root, List<Integer> l) {
   if (root==null)
   return;
        Stack<TreeNode> s = new Stack<TreeNode>();
        s.push(root);
        while(!s.isEmpty()) {
                TreeNode n = s.pop();
                l.add(n.val);
                if (n.right!=null) {
                        s.push(n.right);
                }
                if (n.left != null) {
                        s.push(n.left);
                }
        }
}
```

Question 3:

Description:

Binary tree in-order traversal.



In-order traversal prints: A, B, C, D, E, F, G, H, I.
(source of image:
https://en.wikipedia.org/wiki/Tree_traversal#/medi

Algorithm:

The iterative solution uses stack

1. add the root and all the left nodes
2. pop out a node. Add the right node and all the left side nodes

Code:

```
public List<Integer≥ inorderTraversal(TreeNode root) {
     List<Integer> res = new ArrayList<Integer>();
    if (root==null) return res;
    Stack<TreeNode> s = new Stack<TreeNode>();
    while(root!=null) {
      s.add(root);
      root = root.left;
    }
    while(!s.isEmpty()) {
      TreeNode n = s.pop();
      res.add(n.val);
      if (n.right!= null) {
        n = n.right;
          while(n!=null) {
            s.add(n);
            n= n.left;
          }
      }
    }
    return res;
 }
```

Question 4:

Description:

Implement pre-fix tree (trie). See explanation here: https://en.wikipedia.org/wiki/Trie. Trie is a very useful concept and is a popular component of interview questions.

Algorithm:

Generate the nodes and children recursively.

Code:

```
class TrieNode {
    boolean isWord;
    TrieNode[] children;
    public TrieNode() {
      isWord = false;
      children = 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 node = root;
    int index = 0;
    while(node != null && index <word.length()) {
      char c = word.charAt(index);
      TrieNode[] children = node.children;
      if (children[c-'a']==null) {
        TrieNode next = new TrieNode();
        children[c-'a'] = next;
      }
      node = children[c-'a'];
      if (index==word.length()-1) {
        node.isWord=true;
      }
      index ++;
    }
}

// Returns if the word is in the trie.
public boolean search(String word) {
  if (word==null||word.length()==0) return true;
  TrieNode node = root;
  int index = 0;
  while(node != null && index<word.length()) {
    char c = word.charAt(index);
```

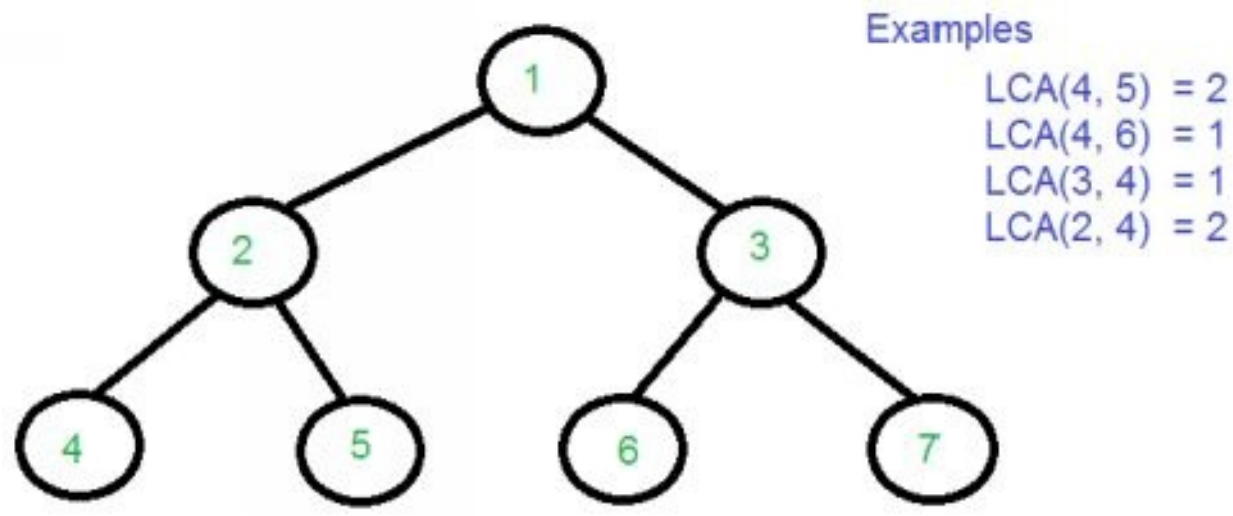
```
        TrieNode[] children = node.children;
        if (children[c-'a']==null) {
          return false;
        } else if (children[c-'a'].isWord==true &&
index==word.length()-1) {
          return true;
        }
        node = children[c-'a'];
        index++;
      }
        return false;
    }


     // Returns if there is any word in the trie
    // that starts with the given prefix.
    public boolean startsWith(String prefix) {
        if (prefix==null||prefix.length()==0) return true;
      TrieNode node = root;
      int index = 0;
      while(node != null && index<prefix.length()) {
        char c = prefix.charAt(index);
        TrieNode[] children = node.children;
        if (children[c-'a']==null) {
          return false;
        }
        node = children[c-'a'];
```

```
            index++;
        }
        return index==prefix.length();
    }
}
```

Question 5:

Description:

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



(source of image: http://d2dskowxfbo68o.cloudfront.net/wp-content/uploads/lca.png)

Algorithm:

Find the lowest node that covers both nodes using recursion.

Code:

```
public TreeNode lowestCommonAncestor(
        TreeNode root, TreeNode p, TreeNode q) {
```

```
        if (root==null) {
                    return null;
        }
        if(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 left;
                else if(left == null && right != null)
                  return right;
                else if(left == null && right == null)
                   return null;
                else
                   return root;
  }
```

Question 6:

Description:

Given a binary search tree and a value, find the node in the tree which is closest to this value.

Algorithm:

The key of the algorithm is that if the node value is smaller than the target value, search for the right subtree. Otherwise, search for the left subtree.

Code:

```
        public int closestValue(TreeNode root, double
target) {
    if (root==null) return -1;
    double min = Double.MAX_VALUE;int num = 0;

    while(root!=null) {
            if (root.val==target) {
                    return root.val;
            } else if (Math.abs(root.val-target)<min) {
                    min = Math.abs(root.val-target);
                    num = root.val;
            }
            if (root.val>target) {
```

```
                root = root.left;
            } else {
                root = root.right;
            }
        }
        return num;
    }
```

Question 7:

Description:

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

Algorithm:

This tutorial is very easy to understand : https://www.youtube.com/watch?v=k2dvEJoHVEM

Code:

```
        public TreeNode buildTree(int[] inorder, int[] postorder) {
                        if (inorder==null||postorder==null||inorder.length==0
                                ||postorder.length==0||inorder.length!=pos
            {
                                return null;
                }
                return build(inorder, postorder, 0,
                        inorder.length-1, 0, postorder.length-1);
            }

    public TreeNode build(int[] inorder, int[] postorder,
```

```
    int is, int ie,   int ps, int pe) {
                        if (is>ie||ps>pe) {
                          return null;
                        }
                        TreeNode current = new
TreeNode(postorder[pe]);

                        int i=is;
                        for (;i<=ie;i++) {
                          if (inorder[i]==postorder[pe]) {
                                  break;
                  }
                        }
                current.left = build(inorder, postorder, is, i-1,
ps, ps+i-is-1);
                current.right = build(inorder, postorder, i+1,
ie, pe-ie+i , pe-1);
                return current;
    }
```

DAY 3 : Linked List

Basics of Linked List

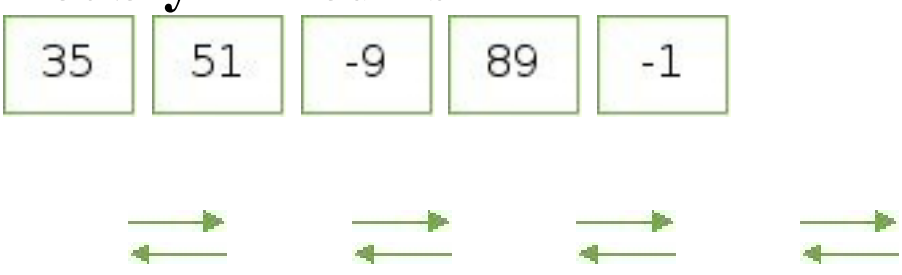
Linked List is a hot topic in coding interviews. Two types are always asked: Singly-linked list and Doubly-linked list. There are many great tutorials on the web so I list some of the greatly resources here:

Singly-linked list

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked Lists/linked lists.html
https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/code/LinkedList.java
http://www.algolist.net/Data_structures/Singly-linked_list

Make sure to practice coding about insert, remove and iterator methods for singly-linked list.

Doubly-linked list



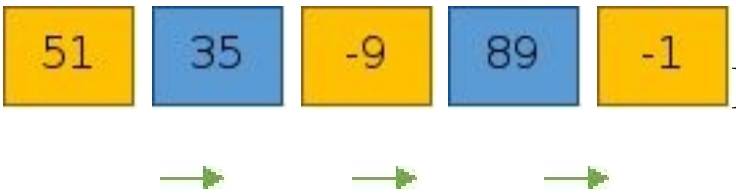
Doubly-linked list is not common in coding questions, except in one very popular question: implement the LRU cache. You will read about this coding question in the next section. For now, let's refresh our memory about the basics by reading the following materials:
http://algs4.cs.princeton.edu/13stacks/DoublyLinke
https://www.youtube.com/watch?v=MZmmSbLJsB4
http://opendatastructures.org/ods-java/3_2_DLList_Doubly_Linked_Li.html

Now you have all the knowledge about linked list to crack the coding questions and let's get started!

Question 1:

Description:

Given a singly linked list, group all odd-index nodes together followed by the even-index nodes. Requirements: 1) the action needs to be in place: the space complexity is O(1) and the time complexity is O(n) (n is the number of nodes) 2) You can not change the value of the node



Before:

After:

Algorithm:

1. consider this list to have 2 head: head1 is for odd-index node and head2 is for even-index node. You can consider this as a list containing odd-index nodes and the other list containing even-index nodes

2. traverse the list to append node: if the node is odd-indexed, append to list 1; otherwise, append to list 2.
3. merge the list

Code:

```
  public ListNode oddEvenList(ListNode head) {
      if (head==null) return head;
      ListNode runner = head, runner1=null, oddTail =
head,
      evenHead = null;

      while(runner!=null) {
         oddTail = runner;
         ListNode next = runner.next;
         // the next node's next
         ListNode next1 = null;
         if (next!=null) {
            next1 = next.next;
            next.next = null;
            if (runner1==null) {
               runner1 = next;
               evenHead = runner1;
            } else {
               runner1.next = next;
               runner1 = runner1.next;
```

```
        }
    }
    runner.next = next1;
    runner = runner.next;
}
// merge the odd list and event list
oddTail.next=evenHead;
return head;
  }
```

Question 2:

Description:

Write method to delete a node in a list (the node is not tail), given access to the node only

Before:

After:

Algorithm:

copy the next node's value to this node, then delete the next node

Code:

```
public void deleteNode(ListNode node) {
    if (node == null || node.next == null) {
                    return;
            }
    node.val = node.next.val;
    node.next = node.next.next;
```

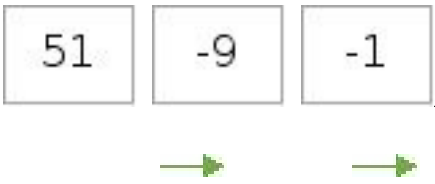
}

Question 3:

Description:

Remove all node that equals to a value in a linked list of all integers

Before:



After:

Algorithm:

1. remember the previous node of the target node, then set previous node's next equal to the target node's next node
2. be careful: if the head of the list equals to the value, it is special case, but we can create a fake head to avoid the special case

Code:

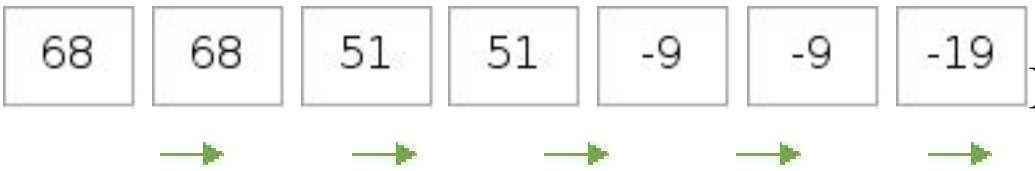
```
public ListNode removeElements(ListNode head, int
```

```
val) {
        ListNode fake = new ListNode(-1);
        fake.next=head;
        ListNode prev= fake;
        while(head!=null) {
                if (head.val == val) {
                        prev.next = head.next;
                        head = prev.next;
                } else {
                        head = head.next;
                        prev=prev.next;
                }
        }
        return fake.next;
    }
```

Question 4:

Description:

Given a **sorted** list, remove the node with duplicated values.



Before:

After:

Algorithm:

traverse the list, compare the node with the next node, if duplicate, remove the next node.

Code:

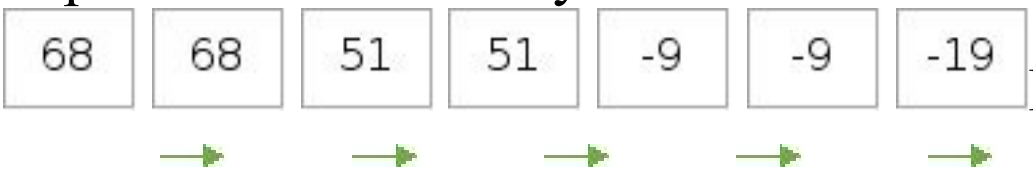
```
public ListNode deleteDuplicates(ListNode head) {
    if (head==null || head.next == null) {
            return head;
    }
    ListNode temp = head;
    while(temp!=null) {
```

```
                ListNode r = temp;
                while(r !=null && r.next!=null &&
r.next.val==r.val) {
                        r= r.next;
                }
                temp.next= r ==null? null: r.next;
                temp = temp.next;
        }
        return head;
    }
```

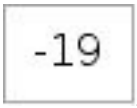
Question 5:

Description:

Given a sorted list, delete all the nodes that have duplicated values. Only distinct values are left.



Before:



After:

Algorithm:

Traverse the list.

Keep comparing the node with the next node until it meets a different value.

If duplicate exists, remove all the duplicated nodes.

Code:

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

```
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        ListNode prev = dummy, current = head;
        while(current!=null) {
                ListNode runner = current.next;
                boolean isUnique = true;
                while(runner!=null&&runner.val ==
current.val) {
                        isUnique = false;
                        runner = runner.next;
                }
                if (!isUnique) {
                        prev.next = runner;
                } else {
                        prev = prev.next;
                }
                current=runner;
        }
        return dummy.next;
 }
```

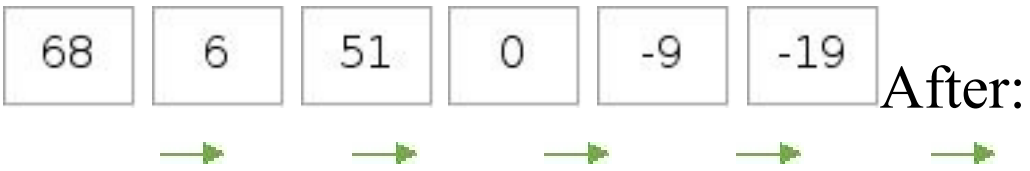
Question 6:

Description:

write a function remove nth node from the end of the list. Do it in one pass

Before:

(n = 3)



After:

Algorithm:

1. create a fast pointer to traverse n nodes one node at a time
2. create another pointer traverse one node at a time together with the fast pointer. When fast pointer goes to the end of the list, this pointer points to the nth node from the end of the list
3. delete the node (the special case is that if the nth node is the head. We can create a fake

head to handle this special case)

Code:

```
    public ListNode removeNthFromEnd(ListNode head,
int n) {
        if (head==null) {
                return head;
        }
        ListNode fake = new ListNode(-1);
        fake.next = head;
        ListNode fast = head, s=head, prev = null;
        while(n>0) {
                fast= fast.next;
                n--;
        }
        while(fast != null) {
                prev= s;
                fast= fast.next;
                s = s.next;
        }

        if (prev !=null) {
                prev.next = s.next;
        } else {
                fake.next = s.next;
        }
```

```
        return fake.next;
    }
```

Question 7:

Description:

Reverse Linked List

Before:

After:

Algorithm:

for each node, reverse itself and the next node.

Code:

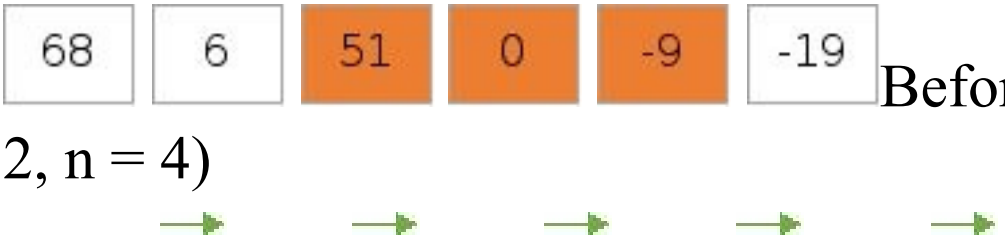
```
public ListNode reverseList(ListNode head) {
  if (head==null) {
            return null;
  }
  ListNode prev = null;
  while(head!=null) {
            ListNode current = head.next;
```

```
                head.next = prev;
                prev = head;
                head = current;
        }
        return prev;
}
```

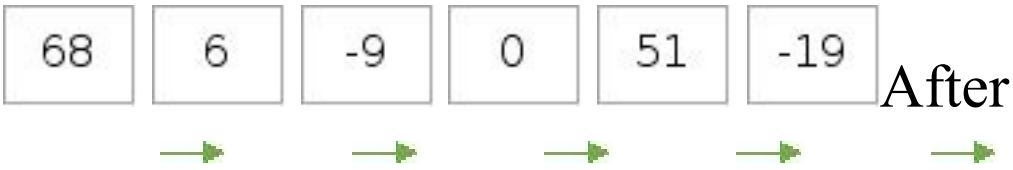
Question 8:

Description:

Reverse the linked list from position m to n in-pass. Do not traverse the list more than once. m and n are within 1 and size of the list and m<=n.



Before: (m = 2, n = 4)



After:

Algorithm:

traverse the list to m-th element. Reverse the element from m to n. The special case is if the m-th element is the head of the list.

Code:

```
  public ListNode reverseBetween(ListNode head, int m,
int n) {
      if (head==null) {
        return head;
```

```
        }
        int index = 0;
        ListNode dummy = new ListNode(0), connectHead
= dummy;
        dummy.next = head;
        ListNode current = head;
        while(current != null && index<m-1) {
          connectHead = current;
          current = current.next;
          index++;
        }
       ListNode start = current;
        while(current != null && index<n-1) {
          current = current.next;
            index++;
        }
        //reverse between current (inclusive) and start
(inclusive)
        ListNode runner = start, next = null, next2 = null,
prev = null;
        ListNode connectTail = current.next;
        current.next = null;
        while(runner != null) {
          next = runner.next;
          if (next != null) {
            next2 = next.next;
            next.next = runner;
```

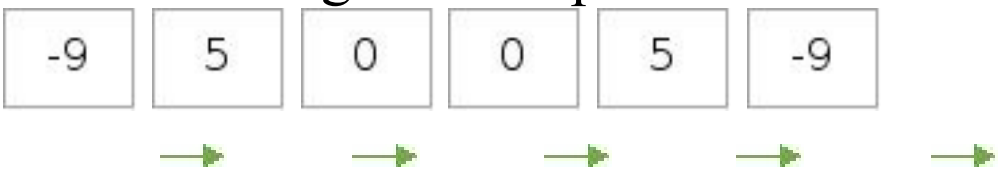
```
        } else {
          next2 = null;
        }
        runner.next = prev;
        prev = next;
        runner = next2;
      }

      connectHead.next = current;
      start.next = connectTail;

      return dummy.next;
  }
```

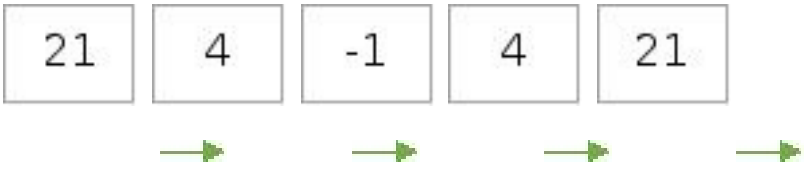
Question 9:

Description:

Write a function to decide whether the linked list is a palindrome

The following lists are palindrome:





This list is not palindrome:

Algorithm:

Revert half of the list.

Compare to see if the nodes are the same.

Pay attention to the case that the middle element is unique (such as the list 1 in the above example)

Code:

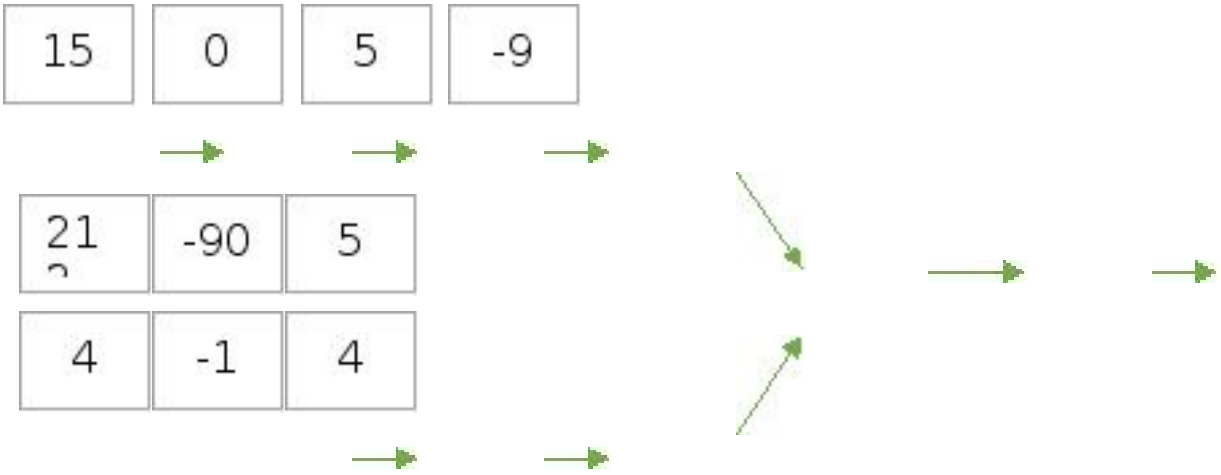
```
  public boolean isPalindrome(ListNode head) {
      if (head == null || head.next == null) {
                  return true;
      }
      boolean isOdd = false;
      ListNode fast = head, mid = head, prev = mid;
                  while(fast != null && fast.next!=null) {
                              isOdd = fast.next.next!=null;
                              fast = fast.next.next;
                              prev=mid;
                              mid = mid.next;
                  }
                  if (!isOdd) {
                     if (prev.val != mid.val) {
                        return false;
                     }
                  }
                  ListNode start1 = revert (mid, head);
                  ListNode start2 = isOdd? mid.next :
mid;
                  while(start1!=null&&start2!=null) {
                              if (start1.val != start2.val) {
                                          return false;
                              }
                              start1= start1.next;
```

```
                start2 = start2.next;
            }
            return start1 == null && start2 == null;
    }

    private ListNode revert(ListNode mid, ListNode head)
{
            ListNode prev = null, next = null;
            while(head != mid) {
                next = head.next;
                head.next = prev;
                prev = head;
                head=next;
            }
            return prev;
    }
```

Question 10:

Description:

Two linked list overlap starting from some nodes. Find the starting node of the overlapping. If the node doesn't exist, return null.



Algorithm:

1. find the length of the 1st list (length1) and 2nd list (length2)
2. have pointer1 pointing to the longer list and advance the | (length1-length2)| steps
3. have pointer 2 pointing to the other list and advance the pointer1 and pointer2 at the same time until the two intersects. The intersecting

node is the node we are trying to find.

Code:

```
 public ListNode getIntersectionNode(ListNode headA,
ListNode headB){
        if (headA == null || headB == null) {
                return null;
        }
        int la = length(headA), lb = length(headB);
        ListNode pa= headA;
        ListNode pb=headB;
        int diff = Math.abs(la - lb);
        int ta=0, tb=0;
        while (la >lb && ta<diff) {
                pa=pa.next;
                ta++;
        }
        while(lb>la && tb<diff) {
                pb=pb.next;
                tb++;
        }
        while(pa!=null && pb!= null && !(pa.equals(pb)))
{
                pa = pa.next;
                pb = pb.next;
        }
        return pa!=null && pa.equals(pb)? pa : null;
```

```
  }
private int length(ListNode l) {
        int count=0;
        ListNode h=l;
        while(h!=null) {
                count++;
                h=h.next;
        }
        return count;
}
```

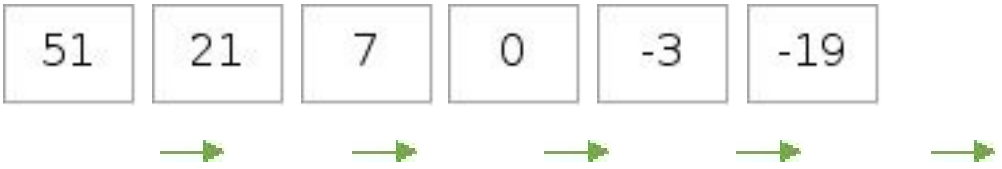
Question 11:

Description:

Sort a singly linked list using a n(log n) algorithm

Before:

After:



Algorithm:

Either merge sort or quick sort. The following code is a merge sort implementation.

Code:

```
public static ListNode sortList(ListNode head) {
                    if (head==null||head.next==null) {
                                return head;
                    }
                    ListNode mid = getMid(head);
                    ListNode right = mid.next;
```

```
                mid.next = null;
                ListNode leftSort = sortList(head);
                ListNode rightSort = sortList(right);
                return merge(leftSort, rightSort);
                    }
        public static ListNode getMid(ListNode head) {
                ListNode l1 = head, l2=head,
prev=head;
                while(l2!=null && l2.next!=null) {
                        l2=l2.next.next;
                        prev = l1;
                        l1=l1.next;

                }
                return prev;
        }
        public static ListNode merge(ListNode node1,
ListNode node2) {
                if (node1==null) {
                        return node2;
                }
                if (node2==null) {
                        return node1;
                }
                ListNode dummy = new ListNode(-1),
current = dummy;
                while(node1!=null ||node2!=null) {
```

```
                    if (node1==null||(node2!=null
&& node1.val>node2.val)) {
                        current.next = node2;
                        node2 = node2.next;
                    } else {
                        current.next = node1;
                        node1 = node1.next;
                    }
                    current = current.next;
                }
                return dummy.next;
            }
```

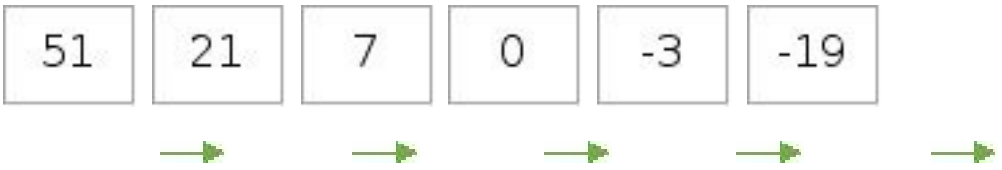
Question 12:

Description:

Sort a linked list using insertion sort

Before:

After:



Algorithm:

you can imagine the list is divided into two ports: sorted part and unsorted part.

In the beginning, the sorted part contains first element of the array and the unsorted part contains the rest.

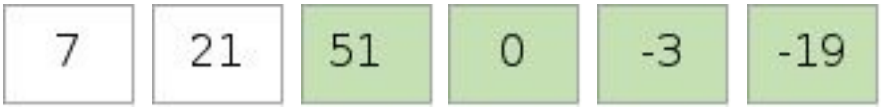
Then for each step, it takes the first element in the unsorted part and inserts it into the right place of the sorted part.

Check node 1:

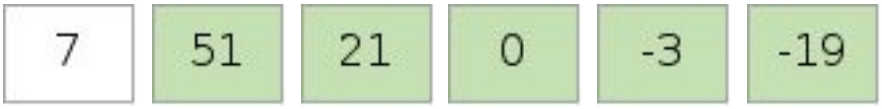
Check node 2:

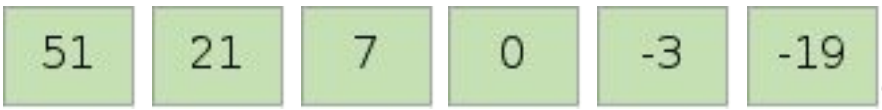
Check node 3:

Check node 4:



Check node 5:





Check node 6:

Code:

```
 public ListNode insertionSortList(ListNode head) {
      if (head==null ||head.next==null) {
         return head;
      }
      ListNode dummy = new ListNode(-1);
                  dummy.next= head;
                  ListNode prev = head, current =
head.next;
                  while(current!=null) {
                        ListNode nextP = current.next;
                        if (current.val < prev.val) {
                              ListNode head1 =
dummy;
                              while
(head1.next!=null &&head1.next.val<current.val) {
                                    head1 =
head1.next;
                              }
                              ListNode next =
head1.next;
```

```
                    head1.next = current;
                    current.next = next;
                    prev.next = nextP;
                } else {
                    prev = current;
                }
                current = nextP;
            }
            return dummy.next;
    }
```

Question 13:

Description:

Given a linked list, detect if there is a cycle

Algorithm:

Create two pointers: pointer 1 advances one step at a time, and pointer 2 advances two steps at a time. If the two pointers meet, there is a cycle

Code:

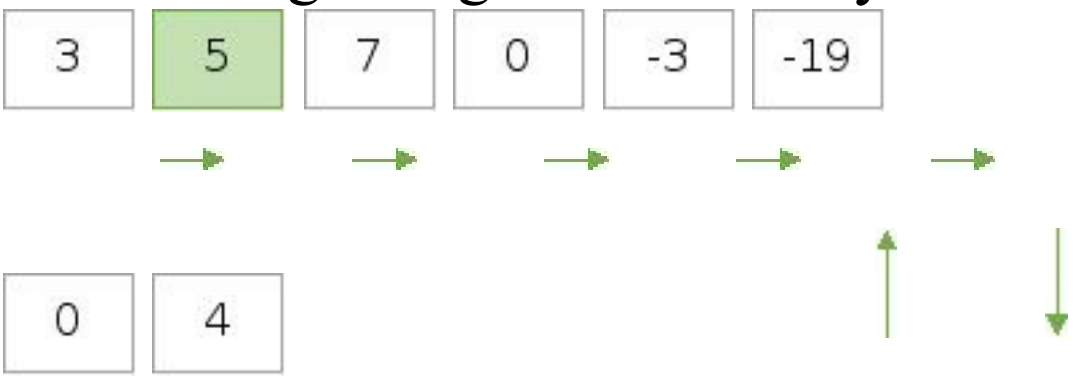
```
public boolean hasCycle(ListNode head) {
        if (head == null) {
                return false;
        }
        ListNode slow = head, fast = head;
        while (fast != null && fast.next != null) {
                fast = fast.next.next;
                slow = slow.next;
```

```
                    if (fast == slow) {
                        return true;
                    }
            }
            return false;
    }
```

Question 14:

Description:

This is a follow up to the previous problem: find the beginning node of the cycle.



Algorithm:

1. use two pointers as in problem 9: pointer 1 advances 1 node at a time and pointer 2 advances 2 nodes at a time until two pointers meet
2. Keep the pointer 1 at the current position, move pointer 2 to the beginning of the list
3. Advance pointer 1 and pointer 2 both one node at a time

4. When the two pointers meet again, the node is the beginning of the loop

Code:

```
public ListNode detectCycle(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;
        while (fast != null && fast.next != null) {
                fast = fast.next.next;
                slow = slow.next;
                if (fast == slow)
                        break;
                }
                if (fast == null || fast.next == null) {
                        return null;
                }
                slow = head;
                while (fast != slow) {
                        fast = fast.next;
                        slow = slow.next;
            }
                return slow;
 }
```

Question 15:

Description:

In a linked list, the node has an additional random pointer that could point to any node in the list or null. Create a deep copy of the list.


```
Class RandomListNode {
        int label;
        RandomListNode next, random;

        RandomListNode(int x) {
                this.label = x;
        }
};
```

Algorithm:

1. Create a hash map to provide a mapping between the node itself and the node its random pointer points to

2. Create a copy of the list without the random pointer
3. Fill in the random pointer for each node

Code:

```
  public RandomListNode
copyRandomList(RandomListNode head) {
        RandomListNode dummy = new
RandomListNode(0), current = dummy;
        HashMap<RandomListNode, RandomListNode>
map =
      new HashMap<RandomListNode,
RandomListNode>();
        RandomListNode runner = head;

        while (runner != null) {
                RandomListNode newNode = new
RandomListNode(runner.label);
                map.put(runner, newNode);
                current.next = newNode;
                current = current.next;
                runner = runner.next;
        }

        runner = head;
        while (runner != null) {
```

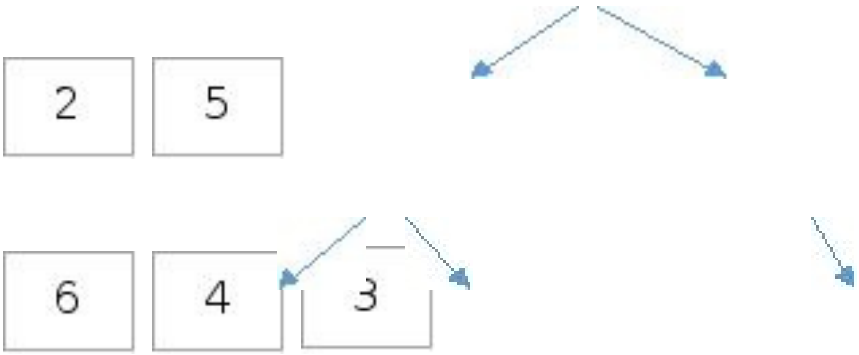
```
                    RandomListNode newNode =
map.get(runner);
                    RandomListNode oldRandom =
runner.random;
                    newNode.random =
map.get(oldRandom);
                    runner = runner.next;
            }

            return dummy.next;
    }
```

Question 16:

Description:

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

The tree:



Convert to the list:

Algorithm:

The key is that each node's right child points to the next node of a pre-order traversal.

Use recursion to convert left sub-tree, right sub-

tree
Link the two parts together.

Code:

```
        public void flatten(TreeNode root) {
                if (root == null || (root.left == null &&
root.right == null)) {
                        return;
                }
                build(root);
        }

        public TreeNode build(TreeNode root) {
                if (root == null || (root.left == null &&
root.right == null)) {
                        return root;
                }
                TreeNode left = build(root.left);
                TreeNode right = build(root.right);
                root.left = null;
                if (left == null) {
                        return root;
                } else {
                        root.right = left;
                        TreeNode runner = left;
                        while (runner.right != null) {
```

```
                runner = runner.right;
            }
            runner.right = right;
        }
        return root;
    }
```

Question 17:

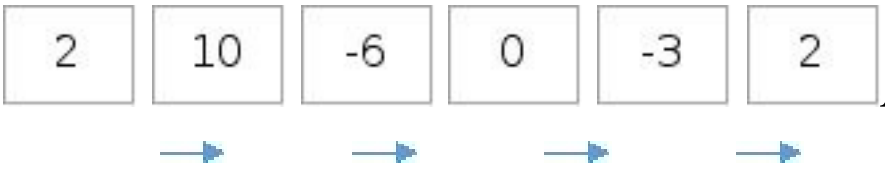
Description:

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

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

Before:

x=3



After:

Algorithm:

1. Imagine creating a new list with nodes which values are greater than or equal to x
2. Traverse the list. When it encounters a node which value is greater than or equal to x, delete from this list and append to the new list
3. Link these two lists together

Code:

```
  public ListNode partition(ListNode head, int x) {
                    if (head == null || head.next == null) {
                              return head;
                    }
                    ListNode dummytail = new
ListNode(-1);
                    ;
                    ListNode dummy = new ListNode(-1);
                    dummy.next = head;
                    ListNode current = head, prev = dummy,
current2 = dummytail;
                    while (current != null) {
                              ListNode next = current.next;
                              if (current.val >= x) {
                                        prev.next = next;
                                        current.next = null;
                                        current2.next =
current;
                                        current2 =
current2.next;
                              } else {
                                        prev = current;
                              }
                              current = next;
```

```
        }
        ListNode runner = dummy;
        while (runner.next != null) {
                runner = runner.next;
        }
        runner.next = dummytail.next;
        return dummy.next;
}
```

Question 18:

Description:

Given a list, rotate it to the right by k elements. K is non-negative and could be bigger than the list size.

Before:

k=2

After:

Algorithm:

Find the Kth element to the right

Cut the list to two parts on this element

Append the second part before the first part

Code:

```
public ListNode rotateRight(ListNode head, int k) {
        if (head == null) {
            return head;
        }
        int size = 1;
```

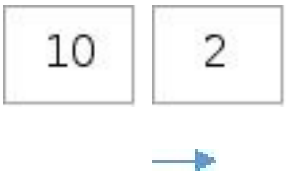
```
        ListNode h = head;
        while (h.next != null) {
                size++;
                h = h.next;
        }
        k = k % size;
        h.next = head;
        for (int i = size - k; i > 1; i--) {
                head = head.next;
        }
        h = head.next;
        head.next = null;
        return h;
}
```

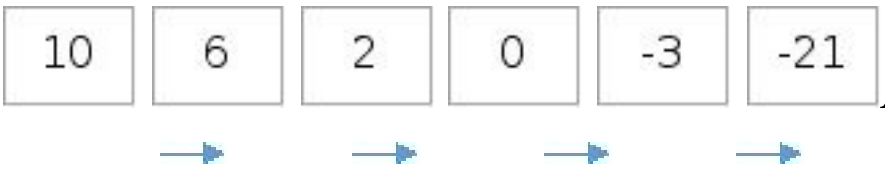
Question 19:

Description:

Given two sorted Linked List, merge them to be one sorted list

Before:





After:

Algorithm:

Create two pointers pointing to the first and second list

Compare the values of the nodes these two pointers point to, take the smaller one and increase the corresponding pointer

If one pointer has traversed its whole list and the other pointer hasn't reached its list, copy over the

remaining nodes.

Code:

```
  public ListNode mergeTwoLists(ListNode l1, ListNode
l2) {
                    if (l1 == null) {
                            return l2;
                    }
                    if (l2 == null) {
                            return l1;
                    }
                    ListNode root = new ListNode(-1);
                    ListNode temp = root;
                    while (l1 != null || l2 != null) {
                            if (l1 != null && (l2 == null ||
l1.val < l2.val)) {
                                    root.next = new
ListNode(l1.val);
                                    l1 = l1.next;
                            } else if (l2 != null && (l1 ==
null || l2.val <= l1.val)) {
                                    root.next = new
ListNode(l2.val);
                                    l2 = l2.next;
                            }
                            root = root.next;
```

```
            }
            return temp.next;
    }
```

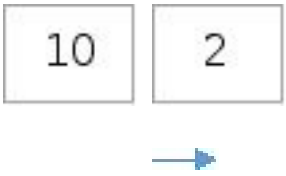
Question 20:

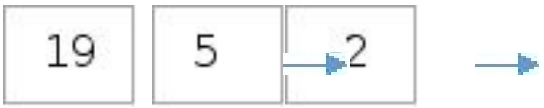
Description:

Given k sorted Linked List, merge them to be one sorted list

Take k = 4 as an example:

Before:





After:

Algorithm:

1. Create a k size heap
2. Put all the first elements of every list into the heap. Pop out the smallest element
3. Insert one element from the same list as the popped element. If no element exist in that list, keep popping the next element
4. Repeat the process until heap is empty

Code:

```
public ListNode mergeKLists(ListNode[] lists) {
                    if (lists == null || lists.length < 1) {
                                return null;
                    }
                    ListNode dummyRoot = new
ListNode(0), runner = dummyRoot;
                    PriorityQueue<ListNode> queue = new
                   PriorityQueue<ListNode>(lists.length,
          new ListNodeComparator());
                    for (ListNode node : lists) {
                                if (node != null)
                                            queue.add(node);
                    }

                    while (!queue.isEmpty()) {
                                ListNode node = queue.poll();
```

```
                runner.next = node;
                if (node.next != null) {
                        queue.add(node.next);
                }
                runner = runner.next;
            }
            return dummyRoot.next;
        }
```

This is the comparator used to compare elements:

```
  class ListNodeComparator implements Comparator<ListNode> {

        @Override
        public int compare(ListNode arg0, ListNode arg1) {
                return arg0.val - arg1.val;
        }
    }
```

Question 21:

LRU cache is a very classic and popular interview question. It stands for Least Recently Used cache. The cache stores key-value pair and supports two operations: Get and Set.

Get() retrieves the value of the key if the key exists in cache and otherwise, return -1;

Set() set the value if the key is not in cache yet. If the key already exists in cache, it will update the value and make it the most recently accessed.

The cache has a capacity. If the cache already reaches the capacity, it will remove the least recently used item before inserting a new item.

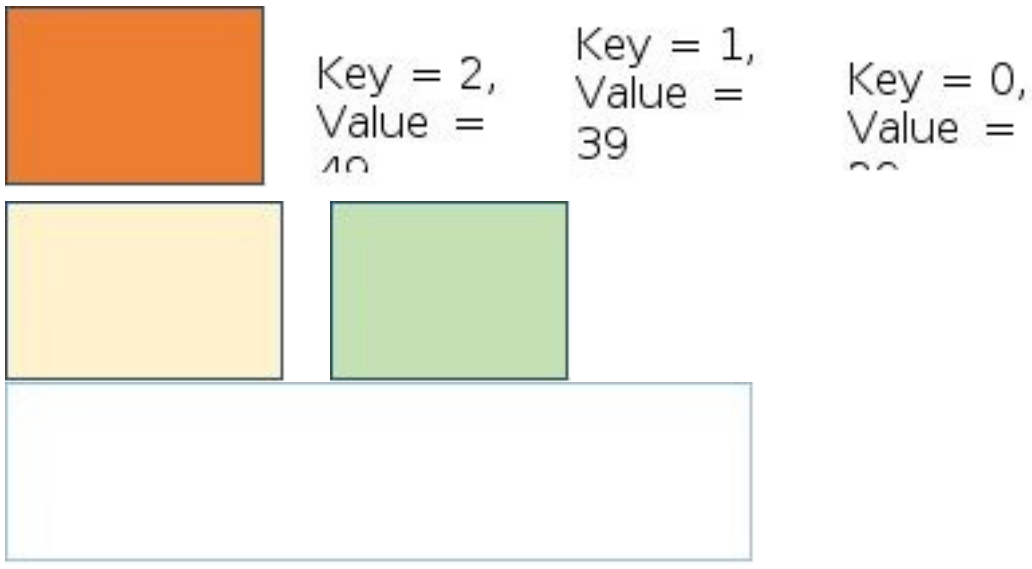
Suppose the LRU has a capacity of 3:

Insert the 1st object (key = 0, value = 29) by calling set (0, 29)

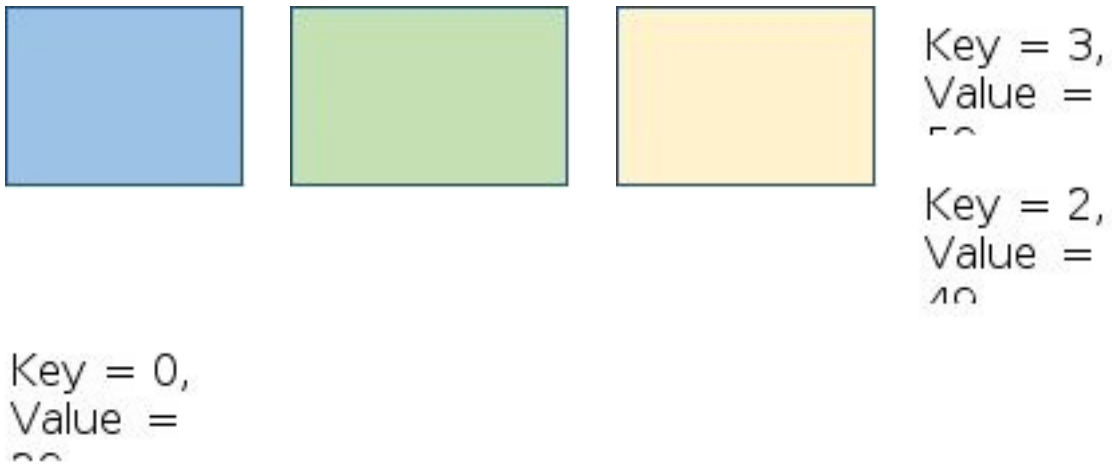
Insert the 2nd object (key = 1, value = 39) by calling set (1, 39)

Call get(0), it returns 29. And we need to move object 0 ahead because it becomes most recently accessed.

Insert the 3rd object (key = 2, value = 49) by calling set (2, 49)



Insert the 4th object (key = 3, value = 59) by calling set (3, 59). It exceeds the capacity of the cache so it drops the least recent used item.



Algorithm:

We use a doubly linked list because it is easy to insert and remove at the beginning and at the end of the list. Then we use a hashmap to do look up in order to get O(1) time complexity.

Code:

```
class ListNode {
    int key;
    int val;
    ListNode prev;
    ListNode next;
```

```
    public ListNode(int k, int v) { key = k; val = v;
}
}

class DoublyLinkedList {
    private ListNode head = null;
    private ListNode tail = null;

    public void addFirst(ListNode node) {
        if (head == null) {
            head = node;
            tail = node;
            return;
        }

        head.prev = node;
        node.next = head;
        node.prev = null;
        head = node;
    }

    public ListNode removeLast() {
        ListNode node = tail;
```

```
    if (tail.prev != null) {
        tail.prev.next = null;
        tail = tail.prev;
    } else {
        head = null;
        tail = null;
    }

    return node;
}

public void promote(ListNode node) {
    if (node.prev == null) {
        return;
    }

    node.prev.next = node.next;
    if (node.next == null) {
        tail = node.prev;
    } else {
        node.next.prev = node.prev;
    }

    head.prev = node;
```

```
        node.next = head;
        node.prev = null;
        head = node;
    }
}

public class LRUCache {
    private final Map<Integer, ListNode>
cachedMap = new HashMap<>();
    private final DoublyLinkedList cachedList =
new DoublyLinkedList();
    private final int capacity;

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

    public int get(int key) {
        if (!cachedMap.containsKey(key)) {
            return -1;
        }

        ListNode targetNode = cachedMap.get(key);
        cachedList.promote(targetNode);
```

```
        return targetNode.val;
    }

    public void set(int key, int value) {
        ListNode targetNode;

        if (cachedMap.containsKey(key)) {
            targetNode = cachedMap.get(key);
            targetNode.val = value;
            cachedList.promote(targetNode);
            return;
        }

        if (cachedMap.size() == capacity) {
            ListNode node = cachedList.removeLast();
            cachedMap.remove(node.key);
        }

        targetNode = new ListNode(key, value);
        cachedList.addFirst(targetNode);
        cachedMap.put(targetNode.key, targetNode);
    }
}
```

DAY 4 : Graph

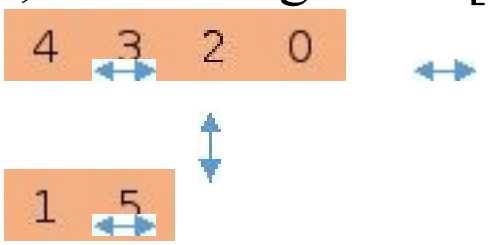
Graph doesn't appear in interview questions as frequently as trees. When they appear, the questions are of few variety so it is relatively easier to prepare for graph based interview questions. Before you start working on any of the interview questions, please make sure you understand the concepts and get familiar with the following algorithms:

1. Union find: https://www.cs.princeton.edu/~rs/AlgsDS07/01
2. Depth first search (DFS): http://www.cs.cornell.edu/courses/cs2112/2012/12sp.html
3. Breath first search (BFS): http://www.cs.cornell.edu/courses/cs2112/2012/12sp.html
4. Topological sort: http://www.cs.cornell.edu/courses/cs2112/2012/12sp.html

Question 1:

Give a list of nodes labeled from 0 to n-1 and a list of undirected edges, find the number of connected components.

For instance, the nodes in the following are 0, 1, 2, 3, 4. The edges are [0, 2], [2, 1], [3, 4], [5, 1]



Algorithm:

This is a union find algorithm. Process the edges one by one to connect nodes. If the nodes couldn't be connected, we consider those as in different components.

Code:

```
public int countComponents(int n, int[][] edges) {
    int count = n;
    int[] roots = new int[n];
    for (int i=0;i<n;i++) {
                        roots[i]=i;
    }
```

```
    for(int[] edge: edges) {
                int n1 = findRoot(roots, edge[0]), n2 =
findRoot(roots, edge[1]);
                if (n1!=n2) {
                        roots[n1]=n2;
                        count--;
                }
    }
    return count;
 }
 public int findRoot(int[] roots, int i) {
            while(roots[i]!=i) {
                    i=roots[i];
            }
            return roots[i];
 }
```

Question 2:

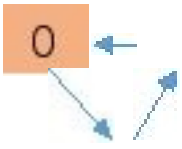
There are n courses you need to take. Some courses have prerequisites as specified in the format of [course, prerequisite]. For instance, [1, 0] means that in order to take course 1, you need to take course 0 first.

If the prerequisites are the following: [1, 0], [0, 2], [2, 1], you know there is no way you can finish the courses because course 1 needs course 0 to be taken, course 0 needs course 2 to be take and course 2 needs course 1 to be taken! It forms a loop and you can not finish the courses.

Given the number of courses and the list of prerequisites, determine whether you can finish taking all the courses.

The following example shows that you can take all the courses.

For the following courses, there is no way to finish them.



3

Algorithm:

This is a typical topological sort problem.

Code:

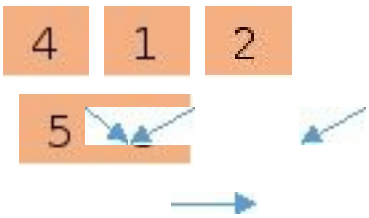
```
public boolean canFinish(int numCourses, int[][]
prerequisites) {
      if
(prerequisites==null||prerequisites.length==0||numCourses
            return true;
       HashMap<Integer, ArrayList<Integer>> map =
            new HashMap<Integer, ArrayList<Integer>>
      ();
       for (int[]d: prerequisites) {
          ArrayList<Integer> required;
          if (map.containsKey(d[0])) {
             required = map.get(d[0]);
          } else {
             required = new ArrayList<Integer>();
          }
          required.add(d[1]);
          map.put(d[0], required);
       }
```

```
    int[] visited = new int[numCourses];
    for (int i=0;i<numCourses; i++) {
        if (visited[i]!=2) {
            if (!dfs(visited, map, i)) return false;
        }
    }
    return true;
}

private boolean dfs(int[] visited, HashMap<Integer,
        ArrayList<Integer>> map, int start) {
    if (visited[start]==1) return false;
    visited[start]=1;
    ArrayList<Integer> requires = map.get(start);
    if (requires != null&& !requires.isEmpty()) {
        for (int r: requires) {
        if (visited[r]==2) continue;
            if (!dfs(visited, map, r)){
                return false;
            }
        }
    }

    visited[start]=2;
    return true;
```

}

Question 3:

This is a follow up from Question 2. Instead of finding out whether you can finish all courses, return the ordering of courses you could take. There might be multiple correct orderings, you only need to return one of them. If there is no way to finish all courses, return an empty array.



Some possible orders are: [2, 1, 3, 4, 5], [1, 2, 3, 4, 5], [4, 2, 1, 3, 5] etc

Algorithm:

This is a typical topological sort problem.

Code:

```
  public int[] findOrder(int numCourses, int[][]
prerequisites) {
        // BFS toposort with Kahn's algorithm
        int[] indegree = new int[numCourses];
        for (int[] p : prerequisites) {
            indegree[p[0]]++;
```

```
        }
        // enqueue vertices with indegree == 0, no
prerequisite is needed
        LinkedList<Integer> queue = new
LinkedList<Integer>();
        for (int i = 0; i < indegree.length; i++) {
            if (indegree[i] == 0) {
                queue.offer(i);
            }
        }
        List<Integer> result = new ArrayList<Integer>();
        while (queue.size() != 0) {
            Integer curr = queue.poll();
            result.add(curr);
            for (int[] p : prerequisites) {
                if (p[1] == curr) {
                    indegree[p[0]]--;
                    if (indegree[p[0]] == 0) {
                        queue.offer(p[0]);
                    }
                }
            }
        }
        if (result.size() != numCourses) {
            return new int[0];
        }
        int[] finalResult = new int[result.size()];
```

```
    for (int i = 0; i < result.size(); i++) {
        finalResult[i] = result.get(i);
    }
    return finalResult;
}
```

Question 4:

Given an undirected graph, return a deep copy of the graph.

Algorithm:

Either DFS or BFS.

Code:

```
public UndirectedGraphNode
cloneGraph(UndirectedGraphNode node) {
    if (node == null) return null;
         Map<UndirectedGraphNode,
UndirectedGraphNode> visited =
        new HashMap<UndirectedGraphNode,
      UndirectedGraphNode>();
    return clone(node, visited);
}

public UndirectedGraphNode
clone(UndirectedGraphNode node,
    Map<UndirectedGraphNode, UndirectedGraphNode>
visited) {
    if (visited.containsKey(node)) {
        return visited.get(node);
```

```
    }
    UndirectedGraphNode copy = new
UndirectedGraphNode(node.label);
    visited.put(node, copy);
    for (UndirectedGraphNode child : node.neighbors) {
      copy.neighbors.add(clone(child, visited));
    }

    return copy;
}
```

Question 5:

"Alien dictionary" is a classic graphql interview question. In an alien language, alphabets are used but the order of letters is unknown. Given a sorted list of words from the alien dictionary ('sorted' means sorted by the rule of this alien language), return the ordering of letters in this language.
If there are multiple orderings, you only need to return one valid ordering.
Given the following sorted list:
["book", "ace", "accurate", "pot", "pocket"]
we can find: 'b' is ahead of 'a', 'a' is ahead of 'p', 'e' is ahead of 'c', 't' is ahead of 'c;
One valid ordering could be 'bapetc'.

Algorithm:

1. Build a graph based on the list of words
2. Topological sort on the graph to get ordering

Code:

```
public String alienOrder(String[] words) {
        String res = "";
    if (words==null||words.length<=0) return res;
```

```
    boolean[][] graph = new boolean[26][26];
    // -1 means the char doesn't exist, 0 means not visited
    int[] tovisit = new int[26];
    Arrays.fill(tovisit, -1);
    for (String s: words) {
      for (char c: s.toCharArray()) {
        graph[c-'a'][c-'a']= true;
        tovisit[c-'a']=0;
      }
    }
    buildGraph(graph, words);
    StringBuilder sb = new StringBuilder();
    if (!topologicalSort(graph, tovisit, sb)) {
        return res;
    }
    return sb.toString();
    }
    public void buildGraph(boolean[][] graph, String[]
words) {
    for (int i=0;i<words.length-1;i++) {
      String w1 = words[i];
      String w2 = words[i+1];
      int len = Math.min(w1.length(), w2.length());
      for (int j=0;j<len;j++) {
        char c1 = w1.charAt(j), c2 = w2.charAt(j);
        if (c1!=c2) {
          graph[c2-'a'][c1-'a']=true;
```

```
          break;
        }
      }
    }

  }

  public boolean topologicalSort(boolean[][] graph, int[]
tovisit, StringBuilder sb) {
    for (int j=0;j<26;j++) {
      // 0 means char exist, but not visited.
      if (tovisit[j]==0) {
        if (!impl(graph, j, tovisit, sb)) {
          return false;
        }
      }
    }
    return true;
  }

  private boolean impl(boolean[][] graph, int i, int[]
visited, StringBuilder sb) {
    if (visited[i]==2)return true;
    //1 means visiting, if it appears again, there is a loop
    if (visited[i]==1) return false;
    visited[i]=1;
    boolean[] requires = graph[i];
```

```
  for (int j=0;j<26;j++) {
    if (requires[j]&&j!=i) {
      if (!impl(graph, j, visited, sb)) {
        return false;
      }
    }
  }
  sb.append((char)(i+'a'));
  visited[i]=2;
  return true;
}
```

Day 5 – Stack & Queue

Question 1:

Given a unix style file path, simplify it. The path is an absolute path.
For instance, '/dir1/"=> "/dir1"
"/dir1/./dir2'/../../dir3/" => "/dir3"

Algorithm:

Put the elements in the stack and follow the linux command rules to pop elements.

Code:

```
  public String simplifyPath(String path) {
      Stack<String> sk = new Stack<String>();
                  String[] strArray = path.split("/");
                  for(int i=0;i<strArray.length;i++){
                        if ((!sk.isEmpty()) &&
(strArray[i].equals(".."))) {
                              sk.pop();
                        }else
if(strArray[i].equals("")||strArray[i].equals(".")||strArray[i]
{
                              //do nothing
                        } else {
```

```
                    sk.push(strArray[i]);
                }
            }
            if (sk.isEmpty()) {
                return "/";
            }
            String result="";

            while(!sk.isEmpty()) {
                result="/"+sk.pop()+result;
            }
            return result;
    }
```

Question 2:

Given an array of integers representing historgram, find the area of the largest rectangle in the histogram. See https://en.wikipedia.org/wiki/Histogram for definition of histogram. Each integer in the area represents a bar that has width 1 and the height of the integer value.

Algorithm:

Use a stack to push and pop value. Use a max variable to keep track of the maximum area.
The key is to find out the left and right indexes of the rectangle.
http://www.geeksforgeeks.org/largest-rectangle-under-histogram/ has a very good explanation of the algorithm.

Code:

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

```
        int max = 0;
        Stack<Integer> stack = new Stack<Integer>();
        for (int i=0;i<heights.length;i++) {
          while(!stack.isEmpty() &&
heights[stack.peek()]>heights[i]) {
            int num = stack.pop();
            int width = !stack.isEmpty() ? i-
stack.peek()-1 : i;
            max = Math.max(max,
heights[num]*width);
          }
          stack.add(i);
        }
        while(!stack.isEmpty()) {
          int num = stack.pop();
          int width = !stack.isEmpty() ? heights.length-
stack.peek()-1 : heights.length;
          max = Math.max(max, heights[num]*width);
        }
        return max;
}
```

Question 3:

Implement queue using stacks. The following methods shall be implemented:

1. Push(): push an element to the back of the queue
2. Pop(): remove the element from the front of the queue
3. Peek(): get the front element
4. Empty(): return whether the queue is empty.

You can only use the standard operation of stack.

Algorithm:

Since stack is 'first in, last out' and queue is 'first in, first out', we can use 2 stacks so we can achieve the order required by the queue.

Code:

```
class MyQueue {
    Stack<Integer> s1 = new Stack<Integer>();
    Stack<Integer> s2 = new Stack<Integer>();
    // Push element x to the back of queue.
    public void push(int x) {
```

```
        while(!s2.isEmpty()) {
            s1.add(s2.pop());
        }
        s1.add(x);
    }

    // Removes the element from in front of queue.
    public void pop() {
        while(!s1.isEmpty()) {
            s2.add(s1.pop());
        }
      s2.pop();
    }

    // Get the front element.
    public int peek() {
         while(!s1.isEmpty()) {
            s2.add(s1.pop());
        }
      return s2.peek();
    }

    // Return whether the queue is empty.
    public boolean empty() {
        return s2.isEmpty() && s1.isEmpty();
    }
}
```

Question 4:

Implement stack using queue. You shall support the following operations:

1) Push(): push the element to the stack
2) Pop(): remove the element on top of the stack

Algorithm:

This question is very similar to Question 3. We need to use 2 queues in order to achieve the stack ordering.

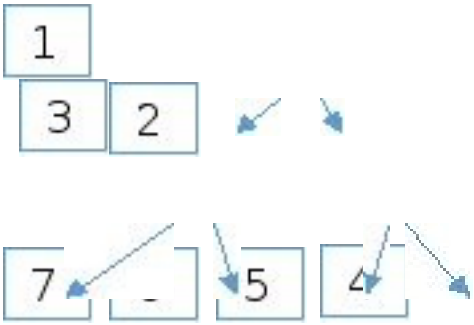
Code:

```
class MyStack {
    private Queue<Integer> q1 = new LinkedList<>();
            private Queue<Integer> q2 = new LinkedList<>
();
            private int top;

            // Push element x onto stack.
            public void push(int x) {
                q1.add(x);
                top = x;
            }
```

```
        // Removes the element on top of the stack.
        public void pop() {
            while (q1.size() > 1) {
                top = q1.remove();
                q2.add(top);
            }
            q1.remove();
            Queue<Integer> temp = q1;
            q1 = q2;
            q2 = temp;
        }
}
```

Question 5:

Given a binary tree, zig-zag the level order traversal of its nodes.



The zig zag level order traversal is :
[1], [3, 2], [4, 5, 6, 7]

Algorithm:

The idea is to use queue (the code implementation uses LinkedList, which is a concrete implementation of queue) to record each level of nodes and a variable to indicate whether to reverse order for the next level.

Code:

```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> res = new
```

```
ArrayList<List<Integer>>();
    if (root==null) {
      return res;
    }
   LinkedList <TreeNode> currentN = new
LinkedList<TreeNode>();
    LinkedList <Integer> current = new LinkedList
<Integer>();
    current.add(root.val);
    currentN.add(root);
    boolean reverse = false;
    while(currentN.size() != 0) {
        LinkedList <TreeNode> nextN = new LinkedList
<TreeNode>();
        LinkedList <Integer> next = new LinkedList
<Integer>();

        for (TreeNode t: currentN) {
          if (t.left!=null)
                    nextN.add(t.left);
          if (t.right!=null)
              nextN.add(t.right);
          if (!reverse) {
             next.add(t.val);
         } else {
           next.add(0, t.val);
         }
```

```
    }
    reverse = !reverse;
    res.add(next);
    currentN = nextN;
  }
  return res;
}
```

Question 6:

Given an array of numbers, verify whether it is preorder traversal of a binary search tree, assuming there is no duplicate in the tree.

Algorithm:

The characteristic of binary search tree is that for each node, its value must be greater than all the values in the left subtree and less than all the values in the right subtree. The idea is traversing the preorder list and uses a stack to store all the previous values. Keep the low value and if there is any violation of the order, return false.

Code:

```
public boolean verifyPreorder(int[] preorder) {
   if (preorder == null|| preorder.length==0)
                        return true;
          Stack<Integer> stack = new Stack<Integer>();
          int low = Integer.MIN_VALUE;
          for(int t: preorder) {
            if (t<low)
               return false;
```

```
            while(!stack.isEmpty() && t>stack.peek()) {
              low = stack.pop();
            }
            stack.push(t);
          }
          return true;
}
```

Question 7:

Evaluate the value of an arithmetic expression in Reverse Polish Notation (https://en.wikipedia.org/wiki/Reverse_Polish_nota
The expression only contains digits and operators '+', '-', '*', '/'.
Example:
["5", "2", "+", "9", "*"] -> ((5+2) * 9)) -> 63

Algorithm:

Use a stack to store numeric values and evaluate the string tokens one by one.
If the token is an operator, pop from the stack to get the numers to apply the operator on.
Push the result into the stack after evaluation.

Code:

```
public int evalRPN(String[] tokens) {
    int res = 0;
    if (tokens==null||tokens.length==0)
        return res;
    Stack<Integer> stack = new Stack<Integer>();
    for (int i=0;i<tokens.length;i++) {
```

```
    String s = tokens[i];
    if (s.equals("+")) {
        int i1 = stack.pop();
        int i2 = stack.pop();
        stack.push(i1+i2);
    } else if (s.equals("-")) {
        int i1 = stack.pop();
        int i2 = stack.pop();
        stack.push(i2-i1);

    } else if (s.equals("*")) {
        int i1 = stack.pop();
        int i2 = stack.pop();
        stack.push(i2*i1);
    }else if (s.equals("/")) {
        int i1 = stack.pop();
        int i2 = stack.pop();
        if (i1==0) {
            System.out.println("couldn't divide 0");
        }
        stack.push(i2/i1);
    } else {
        stack.push(Integer.parseInt(s));
    }
}

return !stack.isEmpty()?  stack.pop() : 0;
```

}

Day 6 – String

Question 1:

Given a roman numeral, convert it to an integer.
See information about roman numeral at
http://literacy.kent.edu/Minigrants/Cinci/romancha

Algorithm:

The key is a mapping between numeral to integer values. With this mapping, calculation is very easy. For special values like 'IV', this formally comes handy: num= num+current-2*prev.

Code:

```
public int romanToInt(String s) {
    if (s==null) {
    return 0;
  }
  int num = 0, prev = Integer.MAX_VALUE;
    for (int i=0;i<s.length();i++) {
      int current = findnum(s.charAt(i));
      if (prev<current) {
        num= num+current-2*prev;
      } else {
        num+=current;
      }
```

```
      prev = current;
    }
  return num;
}
public static int findnum(char c) {
  int number=0;
  switch(c){
      case 'I' : number = 1; break;
      case 'V' : number = 5; break;
      case 'X' : number = 10; break;
      case 'L' : number = 50; break;
      case 'C' : number = 100; break;
      case 'D' : number = 500; break;
      case 'M' : number = 1000; break;
  }
  return number;
}
```

Question 2:

Given a string, determine whether it is a valid panlindrom. See definition of 'palindrome' here: https://en.wikipedia.org/wiki/Palindrome
For this question, you only need to consider alphanumeric characters and you can ignore cases. For instance, "I have : a car ; rac a evah i" is a panlindrom.

Algorithm:

Put 2 pointers at the beginning and end of the string. If the character the pointer points to is not alphanumeric, increase the pointer. Otherwise, see if the values match. If so, increase both pointers until they overlap. If not, return false.

Code:

```
public boolean isPalindrome(String s) {
    if (s==null||s.length()==0) {
      return true;
    }
    s = s.toLowerCase();
    for(int i=0, j=s.length()-1;i<j;) {
```

```
        char c1 = s.charAt(i);
        char c2 = s.charAt(j);
        if (!ischaracter(c1)) {
          i++;
          continue;
        }
        if (!ischaracter(c2)) {
          j--;
          continue;
        }
        if (c1!=c2) return false;
        i++;
        j--;
      }
     return true;
}
private boolean ischaracter(char c) {
  return (c>='a'&&c<='z')|| (c>='0'&&c<='9');
}
```

Question 3:

Given a string, find the longest palindromic substring. See definition of 'palindrome' here: https://en.wikipedia.org/wiki/Palindrome

Algorithm:

There are two formats of palindroms: "abba" and "abcba". Take every character in the string and treat this character as the center and try to form both types of panlindroms. Return the pandlindrom that is the longest.

Code:

```
public String longestPalindrome(String s) {
   if (s==null||s.length()<1) {
         return s;
    }
    String maxS = "";
    int max = 0;
    for (int i=0;i<s.length();i++) {
       //the character is the center of Palindrome
       String max1 = oddUnFold(s, i);
       //the character is one of the centers of
Palindrome(has 2 centers)
```

```
            String max2 = evenUnFold(s, i);
            String win = max1.length()>max2.length() ? max1
: max2;
            if (win.length() > max) {
                max = win.length();
                maxS = win;
            }
        }
        return maxS;
    }
    private static String oddUnFold(String s, int i) {
        int j = i+1, k = i-1;

while(k>=0&&j<s.length()&&s.charAt(j)==s.charAt(k))
{
            j++;
            k--;
        }
        return s.substring(k+1, j);
    }

    private static String evenUnFold(String s, int i) {
        int j = i+1;
        while
(i>=0&&j<s.length()&&s.charAt(j)==s.charAt(i)) {
            j++;
            i--;
```

```
        }
        return s.substring(i+1, j);
    }
}
```

Question 4:

Write a function to return the index of first occurance needle in haystack. If it doesn't exist, return -1.

Algorithm:

The solution is KMP algorithm: https://tekmarathon.com/2013/05/14/algorithm-to-find-substring-in-a-string-kmp-algorithm/

Code:

```
  public int strStr(String haystack, String needle) {
      if (haystack==null||needle==null||haystack.length()
<needle.length())
                              return -1;
      int[] table = buildPrefixTable(needle);
      for (int i=0;i<haystack.length();) {
                              int j=0, k=i;
                              while (k<haystack.length() && j<
needle.length()
                  && haystack.charAt(k)==needle.charAt(j)) {
                                          j++;
                                          k++;
```

```
                    }
                    if (j==needle.length()) {
                            return k-needle.length();
                    } else if (j==0) i++;
                    else {
                            int skip = table[j-1];
                            i= k-skip;
                    }
    }
    return -1;
}
public int[] buildPrefixTable(String s) {
            char[] ptrn = s.toCharArray();
            int i = 0, j = -1;
            int ptrnLen = ptrn.length;
            int[] b = new int[ptrnLen + 1];

            b[i] = j;
            while (i < ptrnLen) {
                  while (j >= 0 && ptrn[i] != ptrn[j]) {
                  j = b[j];
            }
               i++;
               j++;
               b[i] = j;
            }
            return b;
```

```
        }
    }
```

Question 5:

Edit distance problem: given two words, find the minimum number of operations required to convert word 1 to word 2. The valid operations are 1) insert a character 2) delete a character 3) replace a character

Algorithm:

This is a great tutorial about the edit distance problem:
https://web.stanford.edu/class/cs124/lec/med.pdf

Code:

```
public int minDistance(String word1, String word2) {
        if (word1==null&&word2==null) return 0;
                        if (word1==null) return word2.length();
                        if (word2==null) return word1.length();
                        int l1 = word1.length(),
l2=word2.length();
        int[][]dis = new int[l1+1][l2+1];
        for (int i=0;i<=l1;i++) {
                                dis[i][0]=i;
        }
        for (int i=0;i<=l2;i++) {
```

```
                    dis[0][i]=i;
        }
                for (int i=1;i<=l1;i++) {
                    for (int j=1;j<=l2;j++) {
                        char c1 = word1.charAt(i-
1), c2 = word2.charAt(j-1);
                        if (c1==c2)
                            dis[i][j]= dis[i-1]
[j-1];
                        else
                            dis[i][j] = 1+
Math.min(Math.min(dis[i-1][j-1], dis[i-1][j]), dis[i][j-1]);
                    }
                }
                return dis[l1][l2];
    }
}
```

Question 6:

Given an array of strings, group anagrams together. If you don't know what is anagram, check out https://en.wikipedia.org/wiki/Anagram. For instance, if the array input is ["act", "dice", "cat", "iced", "tac"], you shall output: [["act", "tac", "cat"], ["dice", "iced"]].

Algorithm:

1. Use hashmap to store strings with the same 'key' together.
2. Sort all the characters in a word to get a new string as 'key'. For instance, 'act', 'cat' and 'tac' all have key as 'act'.
3. Process all the words so that all the anagrams are grouped together.

Code:

```
 public List<List<String>> groupAnagrams(String[] strs)
{
      List<List<String>> list = new
ArrayList<List<String>>();
```

```
        if (strs==null||strs.length==0) {
                                        return list;
        }
                        HashMap<String, List<String>> m=
new HashMap<String, List<String>>();
                        for(String s: strs) {
                                char[] c= s.toCharArray();
                                Arrays.sort(c);
                                String s1 = new String(c);
                                if (m.containsKey(s1)) {
                                        List<String> l =
m.get(s1);
                                        l.add(s);
                                } else {
                                        List<String> l = new
ArrayList<String>();
                                        l.add(s);
                                        m.put(s1, l);
                                }
                        }
                        for (String k: m.keySet()) {
                                List<String> sortli= m.get(k);
                                Collections.sort(sortli);
                                list.add(sortli);
                        }
                        return list;
    }
```

}

Question 7:

A message is encoded using the following mapping:

‘A’ -> 1

‘B’ -> 2

‘C’ -> 3

…

‘Z’ -> 26.

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

For instance, if the message is ‘12’, it has 2 ways of decoding: ‘AB’ or ‘L’.

Algorithm:

This is a typical dynamic programming problem.

1. Use an array nums[] to store the number of decoding ways. For instance nums[2] is the number of decoding ways for the substring from index 0 to index 2.
2. For each index i, initially, nums[i] = 0
3. Put nums[i] = nums[i-1] because this is one way to decode the string
4. If there is another way to decode (such as

‘12’), consider the latest 2 characters as one letter and add all the ways for the string from index 0 to index i-2. Thus nums[i]+=nums[i-2].

5. If there is a ‘0’ but the previous letter is not ‘1’ or ‘2’, directly return 0 because only 10 and 20 are valid case to decode.

Code:

```
public int numDecodings(String s) {
    if (s==null||s.length()==0||s.charAt(0)=='0') return 0;
    int[] nums = new int[s.length()+1];
    nums[0] = 1;
    nums[1] = 1;
    for (int i=2;i<nums.length;i++) {
      char prev = s.charAt(i-2);
      char c = s.charAt(i-1);
      char next = (i<s.length()) ? s.charAt(i) : 0;
      nums[i]=nums[i-1];
      if (((prev=='1'&& (c>='1'&&c<='9'))
            || (prev=='2' && (c>='1'&&c<='6')))&& next != '0') {
        nums[i]+=nums[i-2];
      } else if (c=='0') {
        if (prev!='1' && prev!='2') return 0;
```

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

Day 7 – Dynamic Programming I

Dynamic programming (DP) interview questions are among the hardest of all interview questions. DP can be applied if the problem has overlapping subproblems and it has optimal substructure.
There are two ways to apply DP: 1) Top-down (Memorization) 2) Bottom-up.
For the interview questions, DP is generally used to avoid unnecessary computation to any problem that involves recursion.
Here are a few very good tutorials:
https://people.cs.clemson.edu/~bcdean/dp_practice
https://www.codechef.com/wiki/tutorial-dynamic-programming
Please make sure that you get familiar with the DP concepts before working on the practice questions.

Question 1:

Given n stairs, you can climb either 1 or 2 steps. How many ways could you climb to the top?

Algorithm:

This is the typical Fibonacci sequence problem.

Code:

```
public int climbStairs(int n) {
    if (n==0||n==1) {
        return 1;
    }
     int[] ways = new int [n+1];
    ways[1]=1;
    ways[2]=2;
    for (int i=3;i<=n;i++) {
                ways[i]=ways[i-1]+ways[i-2];
    }
    return ways[n];
}
```

Question 2:

Given a list of non-negative integers, pick the integers to get the biggest sum and no two adjancent integers could be picked.
Example: the list is [5, 7, 0, 21, 4], then the biggest sum is 28 by picking 7 and 21.

Algorithm:

1. Create an array dp in which dp [i] represents the maximum sum till ith element in the list. Create a max variable to record the maximum sum so far.
2. For any given ith element in the list, since it couldn't pick the i-1th, it could pick any element from index 0 to i-2. So we run a loop to find the largest sum from 0 to i-2th element.
3. Continue this process till the last element. Update max variable whenever the sum at ith element exceeds the previous value of the max variable.

Code:

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

    dp[0]= arr[0];
    int max = dp[0];

    for (int i=1;i<arr.length;i++) {
        int sum = 0;
        if (i==1) sum= arr[i];
        else {
            for (int k = i-2;k>=0;k--) {
                sum = Math.max(sum, arr[i]+dp[k]);
            }
        }
        dp[i]=sum;
        max = Math.max(max, sum);
    }
    return max;
}
```

Question 3:

This is an extension of Question 2. Consider this list is a circular list (meaning if you pick the last element, you couldn't pick the first element and vise versa), how will you modify your code to make it still work?

Algorithm:

Since the first element and last element can not co-exist, we can calculate the maximum sum we can obtain from element 0 to n-1 and from 1 to n. Then we can take a max between the two sums.

Code:

```
public static int rob(int[] nums) {
        if(nums==null||nums.length==0) {
            return 0;
    }
        if (nums.length==1) {
                return nums[0];
        }
        return Math.max(rob(nums, 0, nums.length-2),
        rob(nums, 1, nums.length-1));
}
```

```
public static int rob(int[] nums, int start, int end) {
    int[] arr = new int[nums.length];
    arr[start]=nums[start];int max = arr[start];
    for (int i=start+1;i<=end;i++) {
      int maxsum = 0;
       for (int j=start;j<i-1;j++) {
                    maxsum = Math.max(maxsum, arr[j]);
       }
       arr[i]=maxsum+nums[i];

       max = Math.max(max, arr[i]);
    }
    return max;
}
```

Question 4:

Given a positive integer n, find the least number of perfect square numbers that sum to n.
For instance, for 12, 4+4+4=12, thus return 3.

Algorithm:

1) Create an array to represent the number of square numbers to form 0 to n. array[0] = 0, array[1] = 1.
2) For the number i, it can run a for-loop to find all the number < square root of i and see what are the minimum numbers needed to form i.
3) The last number of the array is the minimum numbers needed to form n.

Code:

```
public static int numSquares(int n) {
    int[] result = new int[n+1];
    result[1]=1;
    for (int i=2;i<=n;i++) {
            int min = result[i-1]+1;
            int nearest = (int)Math.sqrt(i);
```

```
            for (int k=nearest;k>0;k--) {
                int m = result[i-k*k]+1;
                if (m<min) {
                    min=m;
                }
            }
            result[i]=min;
    }
    return result[n];
}
```

Question 5:

Given a money amount and an array of coin denominators, find the fewest number of coins that you need to make up that amount. If the amount couldn't be made up, return -1.

Algorithm:

The algorithm is very similar to question 4.

Code:

```
public int coinChange(int[] coins, int amount) {
    if (coins==null||coins.length==0)
        return 0;
    int[] mins = new int[amount+1];

    for (int i=1;i<=amount;i++) {
        int min = Integer.MAX_VALUE;
        for (int c=0;c<coins.length;c++) {
            if (coins[c]>i)continue;
            int k = 1;
            int sum = 0;
            while(k*coins[c]<=i) {
                if (mins[i-k*coins[c]]==-1) {
                    k++;
```

```
                continue;
            }
            sum= k+mins[i-k*coins[c]];
            k++;
            min= Math.min(sum, min);
        }
    }
    mins[i]=min == Integer.MAX_VALUE ? -1 : min;
  }
  return mins[amount];
}
```

Day 8 – Dynamic Programming II

Question 1:

Find a subarray within an array which has the maximum sum.

For instance, given array [0, -1, 9, 5, 4, -2, 25, -9, 3], the result is [9, 5, 4, -2, 25], which has the maximum sum of all subarries.

Algorithm:

1. Keep a variable sum to track the current sum from left bound to the current index i.
2. If currentSum < 0, it means the left bound shall start from current index I because when the currentSum <0, all numbers before index is is useless. Move left bound to the current index i.
3. Keep track of the maximum sum produced when going through the array.

Code:

```
public int maxSubArray(int[] nums) {
```

```
    int[] sum = new int[nums.length];

    sum[0] = nums[0];
    int max = sum[0];
    for (int i=1;i<nums.length;i++) {
      sum[i] = Math.max(sum[i-1]+nums[i], nums[i]);
      max = Math.max(sum[i], max);
    }
    return max;
}
```

Question 2:

Find a subarray within an array which has the maximum product.
For instance, given array [-1, 5, -6, 7, -9, 0], the result is [5, -6, 7, -9].

Algorithm:

This question is similar to Question 1 but it is harder because the product of 2 negative numbers is positive. Thus we keep two dp arrays: one for recording the current max and the other for recording the current min. The result is the max of the first array.

Code:

```
public int maxProduct(int[] nums) {
    if (nums==null) {
                        return 0;
                }
    int maxHere = nums[0], minHere=nums[0], max=maxHere;
    for (int i=1;i<nums.length;i++) {
            int tmp = maxHere;
```

```
            maxHere = Math.max(nums[i],
                Math.max(minHere*nums[i],
        maxHere*nums[i]));
            minHere = Math.min(nums[i],
                Math.min(minHere*nums[i],
        tmp*nums[i]));

            if (maxHere>max) {
                    max=maxHere;
            }
    }
    return max;
}
```

Question 3:

Given a 2D matrix containing 0s and 1s, find the area of largest square containing all 1s.
For instance, the matrix is:

1	0	0	1	1
1	1	1	1	1
0	1	1	1	0
0	1	1	1	1
1	1	1	1	0

The area of the largest square is 9.

Algorithm:
Create a 2D array ‘grid’ to track the area of the current cell could form. Grid[i][i] means the area with matrix[i][j] in the square. If matrix[i][j] is 0, then grid[i][j] is 0.
We gradually build up the grid by using this formula: grid[i][j] = 1+ Math.min(grid[i-1][j], Math.min(grid[i][j-1], grid[i-1][j-1]));
Track the maximum of the area produced in the grid.

Code:

```
public int maximalSquare(char[][] matrix) {
        if
(matrix==null||matrix.length==0||matrix[0].length==0) {
            return 0;
        }
        int[][] grid = new int[matrix.length]
[matrix[0].length];
        int max=0;
        for (int i=0;i<matrix.length;i++) {
                    for (int j=0;j<matrix[0].length;j++) {
                                if (i == 0 || j == 0) grid[i][j] =
matrix[i][j] - '0';
                                else if (matrix[i][j]=='0') {
                                            grid[i][j]=0;
                                } else {
                                            grid[i][j] = 1+
Math.min(grid[i-1][j], Math.min(grid[i][j-1], grid[i-1][j-
1]));
                                }
                                if (max < grid[i][j]) max = grid[i]
[j];
                    }
        }
        return max*max;
```

```
    }
}
```

Question 4:

Given a 2D matrix, find a path from top left to bottom right which minimizes the sum of all numbers along this path. You can only move down or right at any point.

Algorithm:

Use a 2D matrix to track the minimal sum so far with the current cell added.
The main formular is min[i][j] = grid[i][j]+Math.min(min[i-1][j], min[i][j-1]).
The last value in the matrix is the result.

Code:

```
public int minPathSum(int[][] grid) {
    int[][] min = new int[grid.length][grid[0].length];
            min[0][0] = grid[0][0];
            for (int i=0;i<grid.length;i++) {
                    for (int j=0;j<grid[0].length;j++) {
                            if (i==0&&j==0) {
                                    min[0][0] = grid[0][0];
                            } else {
                                    if (i<1) {
                                            min[i][j] =
```

```
grid[i][j] + min[i][j-1];
                                        }
                                        else if (j<1){
                                                min[i][j] =
grid[i][j] + min[i-1][j];
                                        } else {
                                                min[i][j] =
grid[i][j]+Math.min(min[i-1][j], min[i][j-1]);
                                        }

                                }
                        }
                }
                return min[grid.length-1][grid[0].length-1];
    }
}
```

Question 5:

Given a m*n grid, count the number of ways to travel from the top left to the bottom right. You can only travel either right or down.

Algorithm:

The solution is very similar to Question 3 and Question 4. The key is find the fomular for grid [i] [j], grid[i][j] = grid[i-1][j]+grid[i][j-1];

Code:

```
public int uniquePaths(int m, int n) {
    int[][] grid = new int[m][n];
    grid[0][0] = 1;
    for (int i=0;i<m;i++) {
      grid[i][0] = 1;
    }
    for (int j=0;j<n;j++) {
      grid[0][j] = 1;
    }
    for (int i=1;i<m;i++) {
      for (int j=1;j<n;j++) {
        grid[i][j] = grid[i-1][j]+grid[i][j-1];
      }
```

```
        }
    return grid[m-1][n-1];
}
```

Day 9 – Dynamic Programming III

This section will cover some of the hardest DP questions. With the practice on Day 7 and Day 8, I believe you are fully equipped to solve these!

Question 1:

Given a string containing '(' and ')', find the length of longest valid parentheses substring. For instance, the string is ')(()(())', the length is 4 because '(())' is the longest well-formatted substring.

Algorithm:

Record the starting position of a candidate substring that contains valid paratheses. Update the starting position whenever an invalid parenthese is met.

Code:

```
public int longestValidParentheses(String s) {
    if (s==null||s.length()==0) return 0;
    Stack<Integer> stack = new Stack<Integer>();
    int max = 0, left = 0;
    for (int i=0;i<s.length();i++) {
                        char c = s.charAt(i);
                        if (c=='(') {
```

```
                    stack.add(i);
                } else {
                    if (stack.isEmpty()) {
                        left = i+1;
                    } else {
                        stack.pop();
                        int len =
!stack.isEmpty() ? i-stack.peek(): i-left+1;
                        max =
Math.max(max, len);
                    }
                }
        }
        return max;
    }
```

Question 2:

Paint house problem:

Given a costs matrix n*k, each cell in the matrix represents the cost of painting a house using a color. No adjacent houses have the same color. Find the minimum cost to paint all houses.

For instance,

9	11	3
2	7	22
6	1	2

Costs[0][0] represents the cost of painting house 0 using color 0. Costs[0][1] represents the cost of painting house 0 using color 1… Costs[1][2] represents the cost of painting house 1 using color 2 and so on.

Algorithm:

If there is no constraint, we can choose minimum cost for each house. Since no same color for adjacent houses is allowed, we choose the 2^{nd} min color for the next house. Thus we keep track of the last color used and update the cost.

Code:

```
public int minCost(int[][] costs) {
        int min1=0, min2=0, lastcolor = -1;
    for (int h=0;h<costs.length;h++) {
            int temp1 = Integer.MAX_VALUE,
            temp2 = Integer.MAX_VALUE, cost =0,
     currentColor = -1;
              for (int c=0;c<costs[0].length;c++) {
                      if (c==lastcolor) {
                              cost = min2 +costs[h][c];
                      } else {
                              cost =min1+costs[h][c];
                      }
                      if (cost<temp1) {
                              temp2 = temp1;
                              temp1= cost;
                              currentColor = c;
                      } else if (cost<temp2) {
                              temp2 = cost;
                      }
              }
              min1=temp1;
              min2=temp2;
              lastcolor = currentColor;
     }
```

```
    return min1;
}
```

Question 3:

Given an integer array and each element of the array represents a stock price on a given day, find the maximum profit you can make. You can complete at most two transactions.
For instance,
[9, 18, 7, 0, 4] means for this stock, on day 0, the price is $9. On day 1, the price is $18 and so on. If you buy on day 0 and sell on day 1, this is one transaction that makes $9 profit.

Algorithm:

We use two array to record the maximum profit we could make.
Left[i]: record the max-profit of [0, i] which sells at most at position i.
Right[i]: record the max-profit of [i, len], which buys at most at position i.
Thus the left[i] + right[i] is the profit we made at position i. The maximum sum is the result.

Code:

```
public int maxProfit(int[] prices) {
```

```
    int sum = 0;
    if (prices.length==0) return sum;
    int[] left = new int[prices.length];
    int[] right = new int[prices.length];
    left[0]=0;
    int min= prices[0];
    for (int i=1;i<left.length;i++) {
        left[i]= Math.max(left[i-1], prices[i]-min);
        min = Math.min(min, prices[i]);
    }
    right[prices.length-1]=0;
    int max = prices[prices.length-1];
    for (int j=prices.length-2;j>=0;j--) {
        right[j] = Math.max(right[j+1], max-prices[j]);
        max = Math.max(max, prices[j]);
    }
    for (int i=0;i<prices.length;i++) {
        sum= Math.max(sum, left[i]+right[i]);
    }
    return sum;
}
```

Question 4:

Find the n-th ugly number. See the definition here: http://www.geeksforgeeks.org/ugly-numbers/

Algorithm:

We can use 3 queues to track the numbers. When you take a number from the '2' queue, add number *2, number *3 and number *5 to the queue. When taking from the '3' queue, add number *3, number *5 in the queue (we don't need to add number * 2 because the '2' queue already has it). When taking from the '5' queue, we only need to add number * 5 because the number *2 and number *3 are already in the other two queues.

Code:

```
public int nthUglyNumber(int n) {
  if (n==0) return 0;
   if (n==1) return 1;
   Queue<Long> q2 = new LinkedList<Long>();
   Queue<Long> q3 = new LinkedList<Long>();
   Queue<Long> q5 = new LinkedList<Long>();
```

```
    q2.add(2L);
    q3.add(3L);
    q5.add(5L);
    long res = 0;
    while(n>1) {
      if (q2.peek()<q3.peek()&& q2.peek()<q5.peek()) {
        res = q2.remove();
        q2.add(res*2);
        q3.add(res*3);
        q5.add(res*5);
      } else if (q3.peek()<q5.peek()) {
        res = q3.remove();
        q3.add(res*3);
        q5.add(res*5);
      } else {
        res = q5.remove();
        q5.add(res*5);
      }
      n--;
    }
    return (int)res;
}
```

Question 5:

Given a string s, return the minimum number of cuts that can partition this string to make every substring of the partition a panlindrome.

For example, for “abb”, return 1 because ‘a” and “bb” are panlindroms and this needs 1 cut.

Algorithm:

Preprocess the string by using an matrix to indicate whether the substring from i to j is panlindrom.

Use an array to track the min cut needed at positions. For instance, min[i] means the min cut needed at position i.

Code:

```
public int minCut(String s) {
    if (s==null || s.length()<2) return 0;

    int n = s.length();
    boolean[][] dp = new boolean[n][n];
    for (int i= n-1;i>=0;i--) {
      for (int j= i;j<n;j++) {
        dp[i][j]= (s.charAt(i)==s.charAt(j))&&(j-i<=2||
dp[i+1][j-1]);
      }
    }
    int[] min = new int[s.length()];
    min[0]=0;
```

```
    for (int i=1;i<n;i++) {
      if (dp[0][i]) {
        min[i]=0;
        continue;
      }
      int cut = i;
      for (int j=0;j<=i;j++) {
        if (dp[j][i]) {
          cut = Math.min(cut, min[j-1]+1);
        }
      }
      min[i] = cut;
    }
    return min[n-1];
}
```

Day 10 – Time Complexity Analysis

When you finish your coding or before you start the coding, it is very common for interviewer to ask “What is the time complexity of your algorithm/code ?”. Thus in the last day of this tutorial, let’s study time complexity.

Time complexity in the coding interviews is analyzed by using the Big-O notation. The most straightforward tutorial is https://www.cs.cmu.edu/~adamchik/15-121/lectures/Algorithmic%20Complexity/complex (you shall ignore the ‘Big Omega’ part).

You shall also read the following materials to have a deep understanding:
http://www.geeksforgeeks.org/analysis-of-algorithms-set-1-asymptotic-analysis/
http://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/
http://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/
http://www.geeksforgeeks.org/analysis-of-algorithms-set-4-analysis-of-loops/

http://www.geeksforgeeks.org/analysis-algorithm-set-5-amortized-analysis-introduction/

You will have the skills needed to analyze your algorithms, but the following is a shortcut to give time complexity of common algorithms: http://bigocheatsheet.com/. You can use this cheatsheet to sharpen your skills. For each operation listed in this cheat sheet, you can come up with your own analysis and see if it is correct. When you have gone through the whole process, you shall be very confident to analyze the time complexity of your code/algorithm.

Congratulations! Now you have completely mastered the coding questions. Keep practicing and you will achieve the best performance in the coding interviews!

Some people told me they had great pressure and anxiety in coding interviews, which blocked them from being their best self. I hope to provide a remarkably simple way to liberate yourself from those feelings. My advice is like the following: Do not cling to the outcome and attribute far too much importance to it. Focus on the coding questions instead. Try to be fully engaged in the interaction with interviewers instead of being busy second-guessing yourself.

This is a quotation from the book “Presence:

Bringing Your Boldest Self to Your Biggest

Challenges”:

“Next time you are faced with tense negative moments, imagine approaching it with confidence and excitement instead of doubt and dread. Imaging feeling energized and at ease while you are there, liberated from your fears about how others might be judging you. And imagine leaving it without regret, satisfied that you did your best, regardless of the measurable outcome.”

When you have this mind-set in your coding interview, you will be very successful!