# Notes for Preparing Coding Interview

X Wang

Version 1.0

# CONTENTS

Part I

# FREFACE

Part II

# JAVA QUESTIONS

# 1

## TOP 10 ALGORITHMS FOR CODING INTERVIEW

Web Version, PDF Download Latest Update: 1/9/2014

The following are top 10 algorithms related topics for coding interviews. As understanding those concepts requires much more effort, this list below only serves as an introduction. They are viewed from a Java perspective and the following topics will be covered: String/Array, Linked List, Tree, Graph, Sorting, Recursion vs. Iteration, Dynamic Programming, Bit Manipulation, Probability, Combinations and Permutations, and other problems that need us to find patterns.

### 1.1 STRING/ARRAY

First of all, String in Java is a class that contains a char array and other fields and methods. Without code auto-completion of any IDE, the following methods should be remembered.

```
toCharArray() //get char array of a String
Arrays.sort()  //sort an array
Arrays.toString(char[] a) //convert to string
charAt(int x) //get a char at the specific index
length() //string length
length //array size
substring(int beginIndex)
substring(int beginIndex, int endIndex)
Integer.valueOf()//string to integer
String.valueOf()/integer to string
```

Strings/arrays are easy to understand, but questions related to them often require advanced algorithm to solve, such as dynamic programming, recursion, etc.

Classic problems: Evaluate Reverse Polish Notation, Longest Palindromic Substring, Word Break, Word Ladder, Median of Two Sorted Arrays, Regular Expression Matching, Merge Intervals, Insert Interval, Two Sum, 3Sum, String to Integer, Merge Sorted Array, Valid Parentheses, Implement strStr(), Set Matrix Zeroes.

### 1.2 LINKED LIST

The implementation of a linked list is pretty simple in Java. Each node has a value and a link to next node.

```
class Node {
        int val;
        Node next;

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

Two popular applications of linked list are stack and queue.

Stack

```
class Stack{
        Node top;

        public Node peek(){
                if(top != null){
                        return top;
                }

                return null;
        }

        public Node pop(){
                if(top == null){
                        return null;
                }else{
                        Node temp = new Node(top.val);
                        top = top.next;
                        return temp;
                }
        }

        public void push(Node n){
                if(n != null){
                        n.next = top;
                        top = n;
                }
        }
}
```

Queue

```
class Queue{
        Node first, last;

        public void enqueue(Node n){
                if(first == null){
                        first = n;
                        last = first;
                }else{
                        last.next = n;
```

```
                last = n;
            }
        }

        public Node dequeue(){
            if(first == null){
                return null;
            }else{
                Node temp = new Node(first.val);
                first = first.next;
                return temp;
            }
        }
}
```

It is worth to mention that Java standard library already contains a class called "Stack", and LinkedList can be used as a Queue. (LinkedList implements the Queue interface) If you need a stack or queue to solve problems during your interview, you can directly use them.

Classic Problems: Add Two Numbers, Reorder List, Linked List Cycle, Copy List with Random Pointer, Merge Two Sorted Lists, Remove Duplicates from Sorted List.

## 1.3 TREE

Tree here is normally binary tree. Each node contains a left node and right node like the following:

```
class TreeNode{
        int value;
        TreeNode left;
        TreeNode right;
}
```

Here are some concepts related with trees:

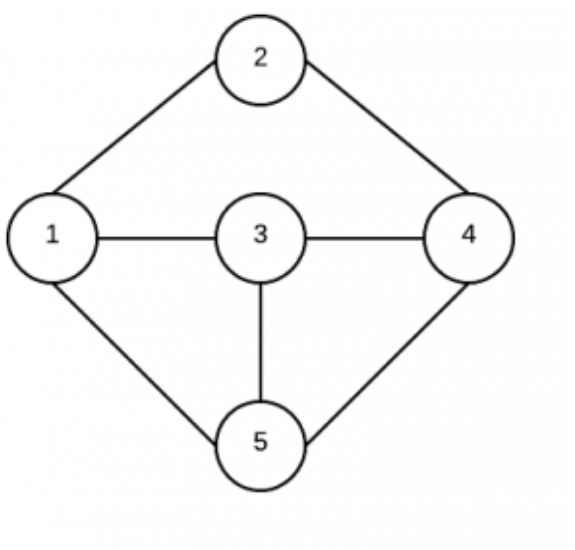
- Binary Search Tree: for all nodes, left children <= current node <= right children
- Balanced vs. Unbalanced: In a balanced tree, the depth of the left and right subtrees of every node differ by 1 or less.
- Full Binary Tree: every node other than the leaves has two children.
- Perfect Binary Tree: a full binary tree in which all leaves are at the same depth or same level, and in which every parent has two children.
- Complete Binary Tree: a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible

Classic problems: Binary Tree Preorder Traversal , Binary Tree Inorder Traversal, Binary Tree Postorder Traversal, Word Ladder, Validate Binary Search Tree, Flatten Binary Tree to Linked List, Path Sum, Construct Binary Tree from Inorder and Postorder Traversal.

## 1.4 GRAPH

Graph related questions mainly focus on depth first search and breath first search. Depth first search is straightforward, you can just loop through neighbors starting from the root node.

Below is a simple implementation of a graph and breath first search. The key is using a queue to store nodes.




1) Define a GraphNode

```
class GraphNode{
        int val;
        GraphNode next;
        GraphNode[] neighbors;
        boolean visited;

        GraphNode(int x) {
                val = x;
        }

        GraphNode(int x, GraphNode[] n){
                val = x;
                neighbors = n;
        }

        public String toString(){
                return "value: "+ this.val;
        }
}
```

2) Define a Queue

```
class Queue{
        GraphNode first, last;
```

```
        public void enqueue(GraphNode n){
                if(first == null){
                        first = n;
                        last = first;
                }else{
                        last.next = n;
                        last = n;
                }
        }

        public GraphNode dequeue(){
                if(first == null){
                        return null;
                }else{
                        GraphNode temp = new GraphNode(first.val, first.
                            neighbors);
                        first = first.next;
                        return temp;
                }
        }
}
```

3) Breath First Search uses a Queue

```
public class GraphTest {

        public static void main(String[] args) {
                GraphNode n1 = new GraphNode(1);
                GraphNode n2 = new GraphNode(2);
                GraphNode n3 = new GraphNode(3);
                GraphNode n4 = new GraphNode(4);
                GraphNode n5 = new GraphNode(5);

                n1.neighbors = new GraphNode[]{n2,n3,n5};
                n2.neighbors = new GraphNode[]{n1,n4};
                n3.neighbors = new GraphNode[]{n1,n4,n5};
                n4.neighbors = new GraphNode[]{n2,n3,n5};
                n5.neighbors = new GraphNode[]{n1,n3,n4};

                breathFirstSearch(n1, 5);
        }

        public static void breathFirstSearch(GraphNode root, int x){
                if(root.val == x)
                        System.out.println("find␣in␣root");

                Queue queue = new Queue();
                root.visited = true;
                queue.enqueue(root);

                while(queue.first != null){
                        GraphNode c = (GraphNode) queue.dequeue();
                        for(GraphNode n: c.neighbors){
```

```
                                if(!n.visited){
                                        System.out.print(n + "␣");
                                        n.visited = true;
                                        if(n.val == x)
                                                System.out.println("Find␣"+n);
                                        queue.enqueue(n);
                                }
                        }
                }
        }
}
```

Output:

value: 2 value: 3 value: 5 Find value: 5 value: 4

Classic Problems: Clone Graph

## 1.5 SORTING

Time complexity of different sorting algorithms. You can go to wiki to see basic idea of them.

* BinSort, Radix Sort and CountSort use different set of assumptions than the rest, and so they are not "general" sorting methods. (Thanks to Fidel for pointing this out)

In addition, here are some implementations/demos: Mergesort, Quicksort, InsertionSort.

## 1.6 RECURSION VS. ITERATION

Recursion should be a built-in thought for programmers. It can be demonstrated by a simple example.

Question:

> *there are n stairs, each time one can climb 1 or 2. How many different ways to climb the stairs?*

Step 1: Finding the relationship before n and n-1.

To get n, there are only two ways, one 1-stair from n-1 or 2-stairs from n-2. If f(n) is the number of ways to climb to n, then f(n) = f(n-1) + f(n-2)

Step 2: Make sure the start condition is correct.

f(0) = 0; f(1) = 1;

```
public static int f(int n){
        if(n <= 2) return n;
        int x = f(n-1) + f(n-2);
        return x;
}
```

The time complexity of the recursive method is exponential to n. There are a lot of redundant computations.

f(5) f(4) + f(3) f(3) + f(2) + f(2) + f(1) f(2) + f(1) + f(2) + f(2) + f(1)

It should be straightforward to convert the recursion to iteration.

```
public static int f(int n) {

        if (n <= 2){
                return n;
        }

        int first = 1, second = 2;
        int third = 0;

        for (int i = 3; i <= n; i++) {
                third = first + second;
                first = second;
                second = third;
        }

        return third;
}
```

For this example, iteration takes less time. You may also want to check out Recursion vs Iteration.

## 1.7 DYNAMIC PROGRAMMING

Dynamic programming is a technique for solving problems with the following properties:

- An instance is solved using the solutions for smaller instances.
- The solution for a smaller instance might be needed multiple times.
- The solutions to smaller instances are stored in a table, so that each smaller instance is solved only once.
- Additional space is used to save time.

The problem of climbing steps perfectly fit those 4 properties. Therefore, it can be solve by using dynamic programming.

```
public static int[] A = new int[100];

public static int f3(int n) {
        if (n <= 2)
                A[n]= n;

        if(A[n] > 0)
                return A[n];
        else
```

```
                A[n] = f3(n-1) + f3(n-2);//store results so only calculate once!
        return A[n];
}
```

Classic problems: Edit Distance, Longest Palindromic Substring, Word Break.

## 1.8 BIT MANIPULATION

Bit operators:

Get bit i for a give number n. (i count from 0 and starts from right)

```
public static boolean getBit(int num, int i){
        int result = num & (1<<i);

        if(result == 0){
                return false;
        }else{
                return true;
        }
}
```

For example, get second bit of number 10.

i=1, n=10 1«1= 10 1010&10=10 10 is not 0, so return true;

Classic Problems: Find Single Number.

## 1.9 PROBABILITY

Solving probability related questions normally requires formatting the problem well. Here is just a simple example of such kind of problems.

> *There are 50 people in a room, what's the probability that two people have the same birthday? (Ignoring the fact of leap year, i.e., 365 day every year)*

Very often calculating probability of something can be converted to calculate the opposite. In this example, we can calculate the probability that all people have unique birthdays. That is: 365/365 * 364/365 * 363/365 * âĂę * 365-n/365 * âĂę * 365-49/365. And the probability that at least two people have the same birthday would be 1 - this value.

```
public static double caculateProbability(int n){
        double x = 1;

        for(int i=0; i<n; i++){
                x *=  (365.0-i)/365.0;
        }

        double pro = Math.round((1-x) * 100);
        return pro/100;
}
```

calculateProbability(50) = 0.97

## 1.10 . COMBINATIONS AND PERMUTATIONS

The difference between combination and permutation is whether order matters.

Example 1:

> *Given 5 numbers - 1, 2, 3, 4 and 5, print out different sequence of the 5 numbers. 4 can not be the third one, 3 and 5 can not be adjacent. How many different combinations?*

Example 2:

> *Given 5 banaba, 4 pear, and 3 apple, assuming one kind of fruit are the same, how many different combinations?*

Please leave your comment if you think any other problem should be here.

## 1.11 . OTHERS

Other problems need us to use observations to form rules to solve them.

Classic problems: Reverse Integer, Pow(x,n).

## 1.12 HIGH FREQUENCY PROBLEMS

1."Valid Number", "Climbing Stairs" Validate Binary Search Tree, Merge Two Sorted Lists, Two Sum, String to Integer, 3Sum, Merge Intervals, Merge Sorted Array, Valid Parentheses, Implement strStr()(), Pow(x,n), Insert Interval, Set Matrix Zeroes

2. "Add Two Numbers", "Binary Tree Traversal(pre/in/post-order)", "Find Single Number", "Word Break", "Reorder List", "Edit Distance", " Reverse Integer", "Copy List with Random Pointer", "Evaluate Reverse Polish Notation", "Word Ladder", "Median of Two Sorted Arrays Java", "Regular Expression Matching",

# 2

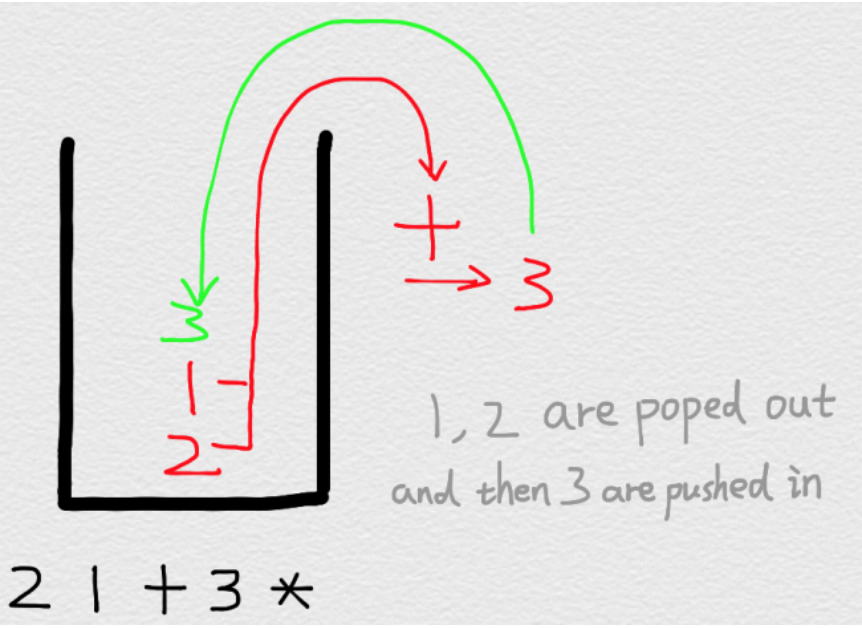
## LEETCODE - EVALUATE REVERSE POLISH NOTATION

The problem:

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

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

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

### 2.1 NAIVE APPROACH

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



The following is the code. It runs great by feeding a small test. However, this code contains compilation errors in leetcode. Why?

```
public class Test {

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

        public static int evalRPN(String[] tokens) {
                int returnValue = 0;
                String operators = "+-*/";

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

                for (String t : tokens) {
                        if (!operators.contains(t)) {
                                stack.push(t);
                        } else {
                                int a = Integer.valueOf(stack.pop());
                                int b = Integer.valueOf(stack.pop());
                                switch (t) {
                                case "+":
                                        stack.push(String.valueOf(a + b));
                                        break;
                                case "-":
                                        stack.push(String.valueOf(b - a));
                                        break;
                                case "*":
                                        stack.push(String.valueOf(a * b));
                                        break;
                                case "/":
                                        stack.push(String.valueOf(b / a));
                                        break;
                                }
                        }
                }

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

                return returnValue;
        }
}
```

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

## 2.2 ACCEPTED SOLUTION

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

```
public class Solution {
    public int evalRPN(String[] tokens) {

        int returnValue = 0;

        String operators = "+-*/";

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

        for(String t : tokens){
            if(!operators.contains(t)){
                stack.push(t);
            }else{
                int a = Integer.valueOf(stack.pop());
                int b = Integer.valueOf(stack.pop());
                int index = operators.indexOf(t);
                switch(index){
                    case 0:
                        stack.push(String.valueOf(a+b));
                        break;
                    case 1:
                        stack.push(String.valueOf(b-a));
                        break;
                    case 2:
                        stack.push(String.valueOf(a*b));
                        break;
                    case 3:
                        stack.push(String.valueOf(b/a));
                        break;
                }
            }
        }

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

        return returnValue;

    }
}
```

# 3

# LEETCODE SOLUTION OF LONGEST PALINDROMIC SUBSTRING IN JAVA

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

## 3.1 NAIVE APPROACH

Naively, we can simply examine every substring and check if it is palindromic. The time complexity is O(n3̂). If this is submitted to LeetCode onlinejudge, an error message will be returned - "Time Limit Exceeded". Therefore, this approach is just a start, we need better algorithm.

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

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

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

        return longestPalindrome;
}

public static boolean isPalindrome(String s) {

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

```
                }
        }

        return true;
}
```

## 3.2 DYNAMIC PROGRAMMING

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

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

Start condition:

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

Changing condition:

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

Time O(n^2) Space O(n^2)

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

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

        int maxLen = 0;
        String longestStr = null;

        int length = s.length();

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

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

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

```
        printTable(table);
        //condition for calculate whole table
        for (int l = 3; l <= length; l++) {
                for (int i = 0; i <= length-l; i++) {
                        int j = i + l - 1;
                        if (s.charAt(i) == s.charAt(j)) {
                                table[i][j] = table[i + 1][j - 1];
                                if (table[i][j] == 1 && l > maxLen)
                                        longestStr = s.substring(i, j + 1);
                        } else {
                                table[i][j] = 0;
                        }
                        printTable(table);
                }
        }

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

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

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

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

## 3.3 SIMPLE ALGORITHM

Time O(n2̂), Space O(1)

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

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

```
        }

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

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

        return longest;
}

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

## 3.4 MANACHER'S ALGORITHM

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

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

# 4

## LEETCODE SOLUTION - WORD BREAK

This is my solution for Word Break in Java.

### 4.1 THE PROBLEM

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

### 4.2 NAIVE APPROACH

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

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

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

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

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

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

        return false;
```

```
    }
}
```

Time: O(2n̂)

## 4.3 DYNAMIC PROGRAMMING

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

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

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

        for(int i=0; i<s.length(); i++){
            //should continue from match position
            if(!t[i])
                continue;

            for(String a: dict){
                int len = a.length();
                int end = i + len;
                if(end > s.length())
                    continue;
                if(s.substring(i, end).equals(a)){
                    t[end] = true;
                }
            }
        }

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

Time: O(string length * dict size)

# 5

## LEETCODE - WORD LADDER

The problem:

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

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

Given:
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",
return its length 5.

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

This problem is a classic problem that has been asked frequently during interviews. The following are two Java solutions.

### 5.1 NAIVE APPROACH

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

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

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

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

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

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

        return len;
    }
}
```

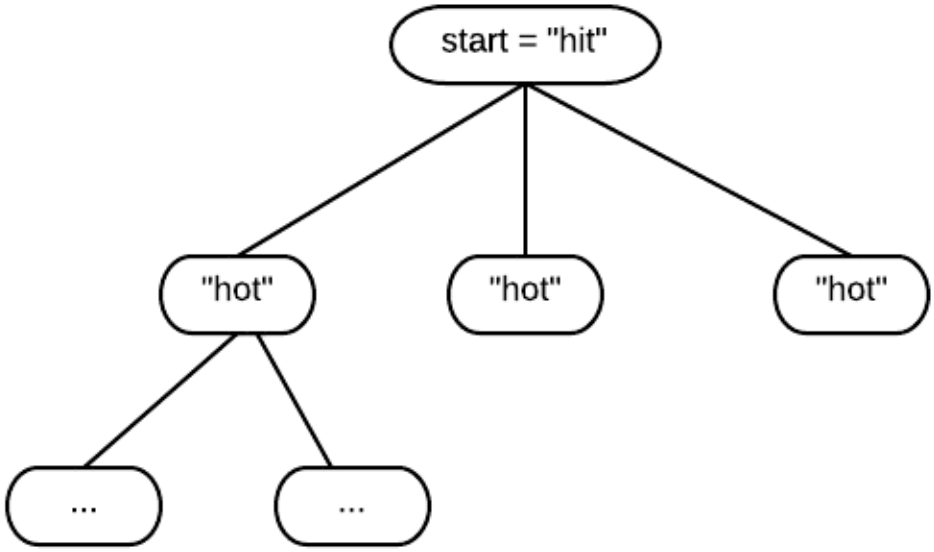
Apparently, this is not good enough. The following example exactly shows the problem. It can not find optimal path. The output is 3, but it actually only takes 2.

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

## 5.2 BREATH FIRST SEARCH

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

Assuming we have all English words in the dictionary, and the start is “hit” as shown in the diagram below.




We can use two queues to traverse the tree, one stores the nodes, the other stores the step numbers. Before starting coding, we can visualize a tree in mind and come up with the following solution.

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

        if (dict.size() == 0)
            return 0;

        int result = 0;

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

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


        while(!wordQueue.isEmpty()){
            String currWord = wordQueue.pop();
            Integer currDistance = distanceQueue.pop();

            if(currWord.equals(end)){
                return currDistance;
            }

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

                    String newWord = new String(currCharArr);
```

```
                    if(dict.contains(newWord)){
                        wordQueue.add(newWord);
                        distanceQueue.add(currDistance+1);
                        dict.remove(newWord);
                    }
                }
            }
        }

        return 0;
    }
}
```

## 5.3 WHAT LEARNED FROM THIS PROBLEM?

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

# 6

## LEETCODE - MEDIAN OF TWO SORTED ARRAYS JAVA

LeetCode Problem:

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

### 6.1 KEYS TO SOLVE THIS PROBLEM

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

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

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

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

        if ((m + n) % 2 != 0) // odd
                return (double) findKth(A, B, (m + n) / 2, 0, m - 1, 0, n - 1);
        else { // even
                return (findKth(A, B, (m + n) / 2, 0, m - 1, 0, n - 1)
                        + findKth(A, B, (m + n) / 2 - 1, 0, m - 1, 0, n - 1)) *
                            0.5;
        }
}

public static int findKth(int A[], int B[], int k,
        int aStart, int aEnd, int bStart, int bEnd) {

        int aLen = aEnd - aStart + 1;
        int bLen = bEnd - bStart + 1;

        // Handle special cases
        if (aLen == 0)
                return B[bStart + k];
        if (bLen == 0)
```

```
                return A[aStart + k];
        if (k == 0)
                return A[aStart] < B[bStart] ? A[aStart] : B[bStart];

        int aMid = aLen * k / (aLen + bLen); // a's middle count
        int bMid = k - aMid - 1; // b's middle count

        // make aMid and bMid to be array index
        aMid = aMid + aStart;
        bMid = bMid + bStart;

        if (A[aMid] > B[bMid]) {
                k = k - (bMid - bStart + 1);
                aEnd = aMid;
                bStart = bMid + 1;
        } else {
                k = k - (aMid - aStart + 1);
                bEnd = bMid;
                aStart = aMid + 1;
        }

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

# 7

## LEETCODE - REGULAR EXPRESSION MATCHING IN JAVA

Problem:

*Implement regular expression matching with support for '.' and '*'.*

```
'.' Matches any single character.
'*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

The function prototype should be:
bool isMatch(const char *s, const char *p)

Some examples:
isMatch("aa","a") return false
isMatch("aa","aa") return true
isMatch("aaa","aa") return false
isMatch("aa", "a*") return true
isMatch("aa", ".*") return true
isMatch("ab", ".*") return true
isMatch("aab", "c*a*b") return true
```

### 7.1 THOUGHTS FOR THIS PROBLEM

Overall, there are 2 different cases: 1) the second char of pattern is "*" , and 2) the second char of pattern is not "*".

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

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

Be careful about the offset.

### 7.2 JAVA SOLUTION

The following Java solution is accepted.

```java
public class Solution {
    public boolean isMatch(String s, String p) {

        if(p.length() == 0)
            return s.length() == 0;

        //p's length 1 is special case
        if(p.length() == 1 || p.charAt(1) != '*'){
            if(s.length() < 1 || (p.charAt(0) != '.' && s.charAt(0) != p.charAt
                (0)))
                return false;
            return isMatch(s.substring(1), p.substring(1));

        }else{
            int len = s.length();

            int i = -1;
            while(i<len && (i < 0 || p.charAt(0) == '.' || p.charAt(0) == s.
                charAt(i))){
                if(isMatch(s.substring(i+1), p.substring(2)))
                    return true;
                i++;
            }
            return false;
        }
    }
}
```

# 8

## LEETCODE - MERGE INTERVALS

Problem:

```
Given a collection of intervals, merge all overlapping intervals.

For example,
Given [1,3],[2,6],[8,10],[15,18],
return [1,6],[8,10],[15,18].
```

### 8.1 THOUGHTS OF THIS PROBLEM

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

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

### 8.2 JAVA SOLUTION

```
class Interval {
        int start;
        int end;

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

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

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

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

```
                return intervals;

            // sort intervals by using self-defined Comparator
            Collections.sort(intervals, new IntervalComparator());

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

            Interval prev = intervals.get(0);
            for (int i = 1; i < intervals.size(); i++) {
                Interval curr = intervals.get(i);

                if (prev.end >= curr.start) {
                    // merged case
                    Interval merged = new Interval(prev.start, Math.
                        max(prev.end, curr.end));
                    prev = merged;
                } else {
                    result.add(prev);
                    prev = curr;
                }
            }

            result.add(prev);

            return result;
        }
}

class IntervalComparator implements Comparator<Interval> {
        public int compare(Interval i1, Interval i2) {
            return i1.start - i2.start;
        }
}
```

# 9

## LEETCODE - INSERT INTERVAL

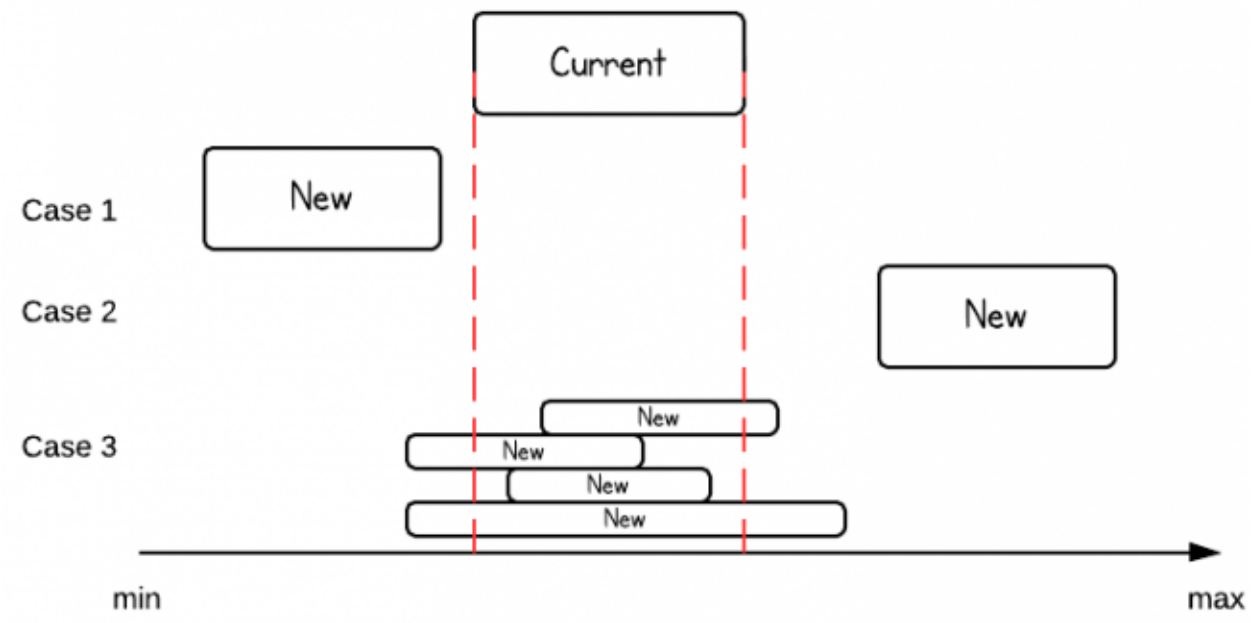
Problem:

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

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

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

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

### 9.1 THOUGHTS OF THIS PROBLEM

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



## 9.2 JAVA SOLUTION

```
/**
 * Definition for an interval.
 * public class Interval {
 *     int start;
 *     int end;
 *     Interval() { start = 0; end = 0; }
 *     Interval(int s, int e) { start = s; end = e; }
 * }
 */
public class Solution {
    public ArrayList<Interval> insert(ArrayList<Interval> intervals, Interval
        newInterval) {

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

        for(Interval interval: intervals){
            if(interval.end < newInterval.start){
                result.add(interval);
            }else if(interval.start > newInterval.end){
                result.add(newInterval);
                newInterval = interval;
            }else if(interval.end >= newInterval.start || interval.start <=
               newInterval.end){
                newInterval = new Interval(Math.min(interval.start, newInterval.
                    start), Math.max(newInterval.end, interval.end));
            }
        }

        result.add(newInterval);

        return result;
    }
}
```

# 10

## LEETCODE - TWO SUM (JAVA)

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

```
Input: numbers={2, 7, 11, 15}, target=9
Output: index1=1, index2=2
```

This problem is pretty straightforward. We can simply examine every possible pair of numbers in this integer array. When the first number is greater than the target, there is no need to find the number number.

Solution:

```
public int[] twoSum(int[] numbers, int target) {
    int[] ret = new int[2];
    for(int i=0; i<numbers.length; i++){
        if(numbers[i] <= target){
            for(int j=i+1; j<numbers.length; j++){
                if(numbers[i] + numbers[j] == target){
                    ret[0]=i+1;
                    ret[1]=j+1;
                }
            }
        }
    }
    return ret;
}
```

# 11

## LEETCODE - 3SUM

Problem:

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

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

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

    A solution set is:
    (-1, 0, 1)
    (-1, -1, 2)
```

### 11.1 NAIVE SOLUTION

Naive solution is 3 loops, and this gives time complexity O(nĝ3). Apparently this is not an acceptable solution, but a discussion can start from here.

```
public class Solution {
    public ArrayList<ArrayList<Integer>> threeSum(int[] num) {
        //sort array
        Arrays.sort(num);

        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer
            >>();
        ArrayList<Integer> each = new ArrayList<Integer>();
        for(int i=0; i<num.length; i++){
            if(num[i] > 0) break;

            for(int j=i+1; j<num.length; j++){
                if(num[i] + num[j] > 0 && num[j] > 0) break;

                for(int k=j+1; k<num.length; k++){
                  if(num[i] + num[j] + num[k] == 0) {

                      each.add(num[i]);
                      each.add(num[j]);
                      each.add(num[k]);
```

```
                    result.add(each);
                    each.clear();
                }
            }
        }
    }

    return result;
  }
}
```

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

Result:

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

## 11.2 BETTER SOLUTION

A better solution is using two pointers instead of one. This makes time complexity of O(n2̂).

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

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

    if (num.length < 3)
        return result;

    // sort array
    Arrays.sort(num);

    for (int i = 0; i < num.length - 2; i++) {
        // //avoid duplicate solutions
        if (i == 0 || num[i] > num[i - 1]) {

            int negate = -num[i];

            int start = i + 1;
            int end = num.length - 1;

            while (start < end) {
                //case 1
                if (num[start] + num[end] == negate) {
                    ArrayList<Integer> temp = new ArrayList<
                        Integer>();
                    temp.add(num[i]);
                    temp.add(num[start]);
                    temp.add(num[end]);
```

```
                                        result.add(temp);
                                        start++;
                                        end--;
                                        //avoid duplicate solutions
                                        while (start < end && num[end] == num[
                                            end + 1])
                                                end--;

                                        while (start < end && num[start] == num[
                                            start - 1])
                                                start++;
                                //case 2
                                } else if (num[start] + num[end] < negate) {
                                        start++;
                                //case 3
                                } else {
                                        end--;
                                }
                        }

                }
        }

        return result;
}
```

# 12

## LEETCODE - STRING TO INTEGER (ATOI)

Problem:

```
Implement atoi to convert a string to an integer.

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

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

### 12.1 THOUGHTS FOR THIS PROBLEM

The vague description give us space to consider different cases.

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

### 12.2 JAVA SOLUTION

```
public int atoi(String str) {
        if(str == null || str.length() < 1)
        return 0;

        //trim white spaces
        str = str.trim();

        char flag = '+';

        //check negative or positive
        int i=0;
        if(str.charAt(0) == '-'){
        flag = '-';
        i++;
```

```
        }else if(str.charAt(0) == '+'){
                i++;
    }
        //use double to store result
        double result = 0;

        //calcualte value
        while(str.length() > i && str.charAt(i) >= '0' && str.charAt(i) <= '9'){
                result = result * 10 + (str.charAt(i)-'0');
                i++;
        }

        if(flag == '-')
                result = -result;

        //handle max and min
        if(result > Integer.MAX_VALUE)
                return Integer.MAX_VALUE;

        if(result < Integer.MIN_VALUE)
                return Integer.MIN_VALUE;

        return (int)result;
}
```

# 13

## LEETCODE - MERGE SORTED ARRAY (JAVA)

Problem:

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

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

### 13.1 THOUGHTS FOR THIS PROBLEM

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

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

### 13.2 JAVA SOLUTION 1

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

        while(m > 0 && n > 0){
            if(A[m-1] > B[n-1]){
                A[m+n-1] = A[m-1];
                m--;
            }else{
                A[m+n-1] = B[n-1];
                n--;
            }
        }

        while(n > 0){
            A[m+n-1] = B[n-1];
            n--;
        }
    }
}
```

## 13.3 JAVA SOLUTION 2

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

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

        while (k >= 0) {
                if (j < 0 || (i >= 0 && A[i] > B[j]))
                        A[k--] = A[i--];
                else
                        A[k--] = B[j--];
        }
}
```

# 14

## LEETCODE - VALID PARENTHESES (JAVA)

Problem:

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

### 14.1 THOUGHTS ABOUT THIS PROBLEM

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

### 14.2 JAVA SOLUTION

```
public static boolean isValid(String s) {
        HashMap<Character, Character> map = new HashMap<Character, Character>();
        map.put('(', ')');
        map.put('[', ']');
        map.put('{', '}');

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

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

                if (map.keySet().contains(curr)) {
                        stack.push(curr);
                } else if (map.values().contains(curr)) {
                        if (!stack.empty() && map.get(stack.peek()) == curr) {
                                stack.pop();
                        } else {
                                return false;
                        }
                }
        }

        return stack.empty();
}
```

## 14.3 SIMPLIFIED JAVA SOLUTION

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

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

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

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

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

# 15

## LEETCODE - IMPLEMENT STRSTR() (JAVA)

Problem:

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

### 15.1 THOUGHTS

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

### 15.2 JAVA SOLUTION

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

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

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

        if (needleLen == 0)
                return haystack;

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

                int k = i;
                int j = 0;

                while (j < needleLen && k < haystackLen && needle.charAt(j) ==
                    haystack.charAt(k)) {
                        j++;
                        k++;
                        if (j == needleLen)
```

```
                    return haystack.substring(i);
                }

            }

            return null;
}
```

# 16

## LEETCODE - SET MATRIX ZEROES (JAVA)

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

### 16.1 THOUGHTS ABOUT THIS PROBLEM

This problem can solve by following 4 steps:

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

### 16.2 JAVA SOLUTION

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

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

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

        //mark zeros on first row and column
```

```
        for(int i=1; i<matrix.length; i++){
            for(int j=1; j<matrix[0].length; j++){
                if(matrix[i][j] == 0){
                    matrix[i][0] = 0;
                    matrix[0][j] = 0;
                }
            }
        }

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

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

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

    }
}
```

# 17

## LEETCODE SOLUTION - ADD TWO NUMBERS IN JAVA

The problem from Leetcode.

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

This is a simple problem. Just follow the steps like the following:

- A flag to mark if previous sum is >= 10
- Handle the situation that one list is longer than the other
- Correctly move the 3 pointers p1, p2, and p3
- Handle the situation of 5 + 5

### 17.1 SOLUTION 1

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

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

        ListNode p1 = l1;
        ListNode p2 = l2;

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

```
int val;//store sum

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

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

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

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

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

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

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

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

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

        if(flag){
            val = p1.val + 1;
            if(val >= 10){
                flag = true;
```

```
                    }else{
                        flag = false;
                    }
                }else{
                    val = p1.val;
                    flag = false;
                }

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

            p3 = p3.next;
        }

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

        return newHead.next;
    }
}
```

## 17.2 SOLUTION 2

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

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

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

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

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

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

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

        return newHead.next;
    }
}
```

Exactly the same thing!

# 18

## REORDER LIST IN JAVA

This article shows a Java solution for solving the reorder list problem from Leetcode: Reorder List.

> *Given a singly linked list L: Lo→L1→âĂę→Ln-1→Ln, reorder it to: Lo→Ln→L1→Ln-1→L2→Ln-2→âĂę You must do this in-place without altering the nodes' values. For example, Given 1,2,3,4, reorder it to 1,4,2,3.*

It is not straightforward because it requires "in-place" operations.

### 18.1 SOLUTION

There are three key steps to solve this problem:

- Break list in the middle to two lists
- Reverse the order of the second list
- Merge two list back together

The following code is a complete runnable class with testing.

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

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

public class ReorderList {

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

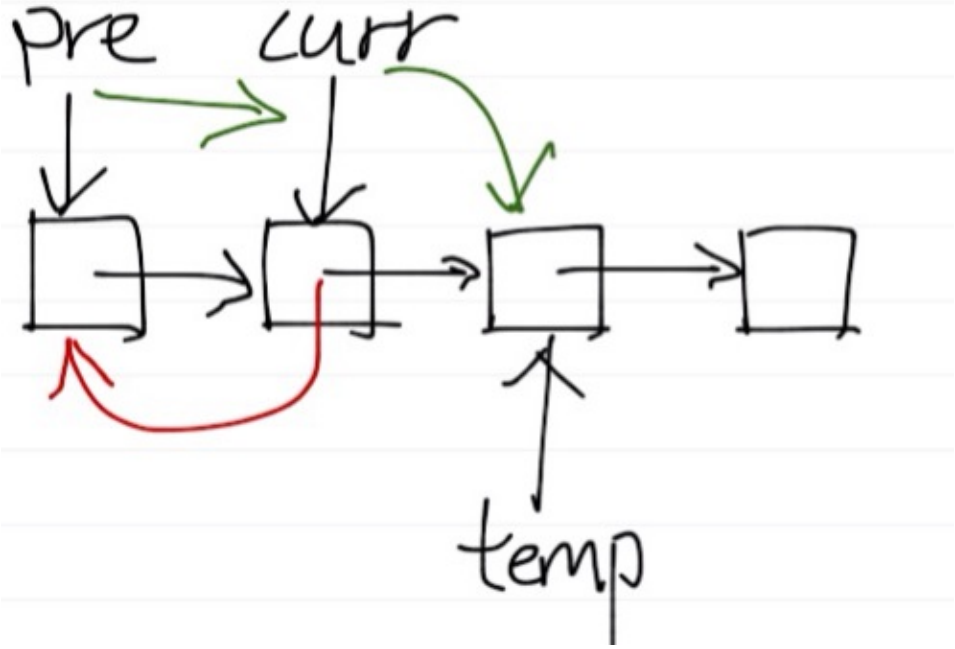
```
        n1.next = n2;
        n2.next = n3;
        n3.next = n4;

        printList(n1);

        reorderList(n1);

        printList(n1);
    }

    public static void reorderList(ListNode head) {

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

            ListNode slow = head;
            ListNode fast = head;

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

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

            // now should have two lists: head and fast

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

            ListNode p1 = head;
            ListNode p2 = second;

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

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

                p1 = temp1;
                p2 = temp2;
```

```
                        }
                }
        }

        public static ListNode reverseOrder(ListNode head) {

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

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

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

                head.next = null;

                return pre;
        }

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

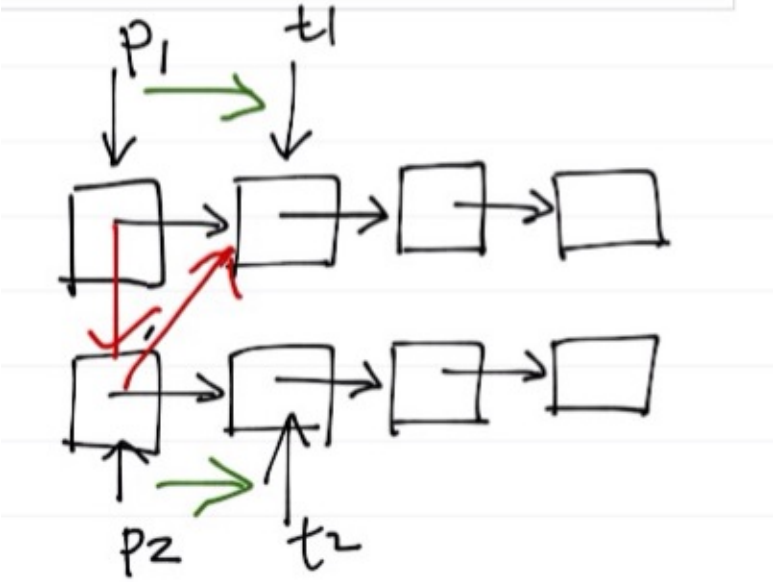
## 18.2 TAKEAWAY MESSAGE FROM THIS PROBLEM

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

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

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

head.next = null;
```



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

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

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

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



# 19

## LEETCODE - LINKED LIST CYCLE

Leetcode Problem: Linked List Cycle

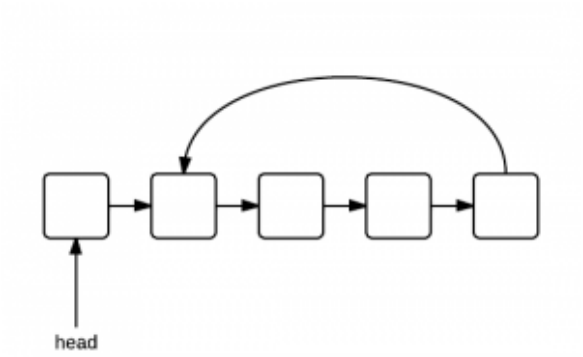
*Given a linked list, determine if it has a cycle in it.*

### 19.1 NAIVE APPROACH

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

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

        if(head == null)
            return false;

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

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

        return false;
    }
}
```

Result:

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




## 19.2 ACCEPTED APPROACH

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

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

        if(head == null)
            return false;

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

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

            if(slow == fast)
                return true;
        }

        return false;
    }
}
```

# 20

## LEETCODE - COPY LIST WITH RANDOM POINTER

Problem:

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

### 20.1 HOW TO SOLVE THIS PROBLEM?

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

### 20.2 FIRST ATTEMPT

What is wrong with the following code?

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

        if(head == null)
            return null;

        RandomListNode p = head;

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

```
            p = copy.next;
        }

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

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

        return head.next;
    }
}
```

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

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

## 20.3 CORRECT SOLUTION

```
public RandomListNode copyRandomList(RandomListNode head) {

        if (head == null)
                return null;

        RandomListNode p = head;

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

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

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

        return newHead;
}
```

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

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

# 21

## LEETCODE - MERGE TWO SORTED LISTS (JAVA)

Problem:

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

### 21.1 KEY TO SOLVE THIS PROBLEM

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

### 21.2 JAVA SOLUTION

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

        ListNode p1 = l1;
        ListNode p2 = l2;

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

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

```
            }else{
                p.next = p2;
                p2 = p2.next;
            }

            p = p.next;
        }

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

        return fakeHead.next;
    }
}
```

# 22

## LEETCODE - REMOVE DUPLICATES FROM SORTED LIST

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

For example,

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

### 22.1 THOUGHTS

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

### 22.2 SOLUTION 1

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

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

        while(p != null){
            if(p.val == prev.val){
                prev.next = p.next;
                p = p.next;
```

```
                //no change prev
            }else{
                prev = p;
                p = p.next;
            }
        }

        return head;
    }
}
```

## 22.3 SOLUTION 2

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

        ListNode p = head;

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

        return head;
    }
}
```

# 23

## LEETCODE SOLUTION FOR BINARY TREE PREORDER TRAVERSAL IN JAVA

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

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

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

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

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

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

        if(root == null)
            return returnList;

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

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

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

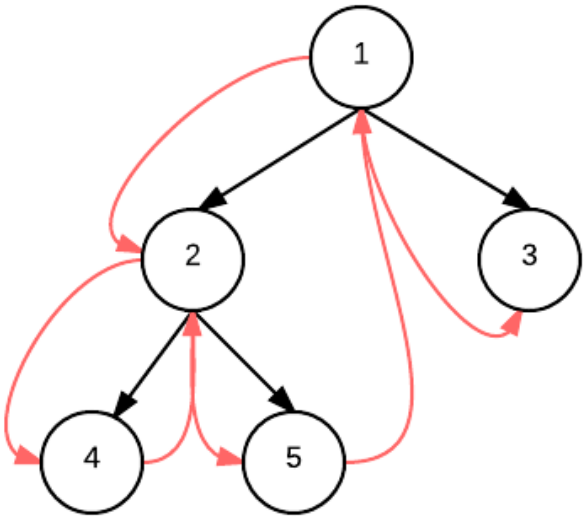
```
        }
        return returnList;
    }
}
```

# 24

## LEETCODE SOLUTION OF BINARY TREE INORDER TRAVERSAL IN JAVA

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

- The order of "inorder" is: left child ->parent ->right child
- Use a stack to track nodes
- Understand when to push node into the stack and when to pop node out of the stack




```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ArrayList<Integer> inorderTraversal(TreeNode root) {
        // IMPORTANT: Please reset any member data you declared, as
        // the same Solution instance will be reused for each test case.
         ArrayList<Integer> lst = new ArrayList<Integer>();
```

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

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

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

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

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

        return lst;
    }
}
```

# 25

## LEETCODE SOLUTION OF ITERATIVE BINARY TREE POSTORDER TRAVERSAL IN JAVA

The key to to iterative postorder traversal is the following:

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

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

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ArrayList<Integer> postorderTraversal(TreeNode root) {

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

        if(root == null)
            return lst;

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

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

            // go down the tree.
            //check if current node is leaf, if so, process it and pop stack,
            //otherwise, keep going down
            if(prev == null || prev.left == curr || prev.right == curr){
```

```
                //prev == null is the situation for the root node
                if(curr.left != null){
                    stack.push(curr.left);
                }else if(curr.right != null){
                    stack.push(curr.right);
                }else{
                    stack.pop();
                    lst.add(curr.val);
                }

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

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

            prev = curr;
        }

        return lst;
    }
}
```

# 26

## LEETCODE - WORD LADDER

The problem:

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

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

Given:
start = "hit"
end = "cog"
dict = ["hot","dot","dog","lot","log"]
As one shortest transformation is "hit" -> "hot" -> "dot" -> "dog" -> "cog",
return its length 5.

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

This problem is a classic problem that has been asked frequently during interviews. The following are two Java solutions.

### 26.1 NAIVE APPROACH

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

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

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

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

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

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

        return len;
    }
}
```

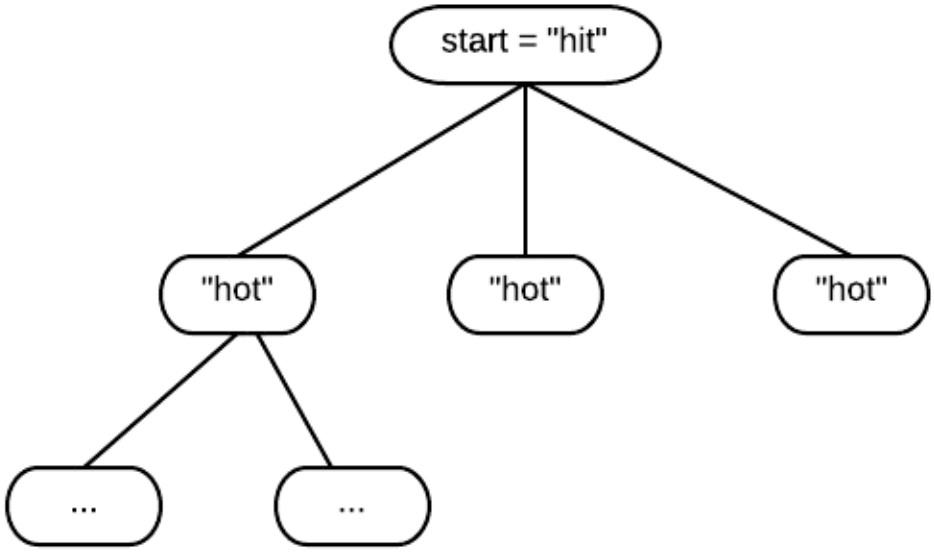
Apparently, this is not good enough. The following example exactly shows the problem. It can not find optimal path. The output is 3, but it actually only takes 2.

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

## 26.2 BREATH FIRST SEARCH

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

Assuming we have all English words in the dictionary, and the start is “hit” as shown in the diagram below.




We can use two queues to traverse the tree, one stores the nodes, the other stores the step numbers. Before starting coding, we can visualize a tree in mind and come up with the following solution.

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

        if (dict.size() == 0)
            return 0;

        int result = 0;

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

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


        while(!wordQueue.isEmpty()){
            String currWord = wordQueue.pop();
            Integer currDistance = distanceQueue.pop();

            if(currWord.equals(end)){
                return currDistance;
            }

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

                    String newWord = new String(currCharArr);
```

```
                    if(dict.contains(newWord)){
                        wordQueue.add(newWord);
                        distanceQueue.add(currDistance+1);
                        dict.remove(newWord);
                    }
                }
            }
        }

        return 0;
    }
}
```

## 26.3 WHAT LEARNED FROM THIS PROBLEM?

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

# 27

## LEETCODE - VALIDATE BINARY SEARCH TREE (JAVA)

Problem:

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

Assume a BST is defined as follows:

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

### 27.1 THOUGHTS ABOUT THIS PROBLEM

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

### 27.2 JAVA SOLUTION

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

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

public class Solution {

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

        public static boolean validate(TreeNode root, int min, int max) {
```

```
                if (root == null) {
                        return true;
                }

                // not in range
                if (root.val <= min || root.val >= max) {
                        return false;
                }

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

# 28

## LEETCODE - FLATTEN BINARY TREE TO LINKED LIST

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

For example, Given

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

The flattened tree should look like:

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

### 28.1 THOUGHTS

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

### 28.2 JAVA SOLUTION

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

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

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

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

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

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

# 29

## LEETCODE - PATH SUM

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

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

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

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

### 29.1 THOUGHTS

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

For example above, the queue would be: 5 - 4 - 8 - 11 - 13 - 4 - 7 - 2 - 1. It will check node 13, 7, 2 and 1.

This is a typical breadth first search(BFS) problem.

### 29.2 JAVA SOLUTION

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

```
    public boolean hasPathSum(TreeNode root, int sum) {
        if(root == null) return false;

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

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

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

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

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

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

        return false;
    }
}
```

# 30

## CONSTRUCT BINARY TREE FROM INORDER AND POSTORDER TRAVERSAL

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

### 30.1 THROUGHTS

This problem can be illustrated by using a simple example.

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

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

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

### 30.2 JAVA SOLUTION

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

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

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

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

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

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

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

        return root;
    }
}
```

# 31

## LEETCODE CLONE GRAPH JAVA

LeetCode Problem:

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

**OJ's undirected graph serialization:**

Nodes are labeled uniquely.

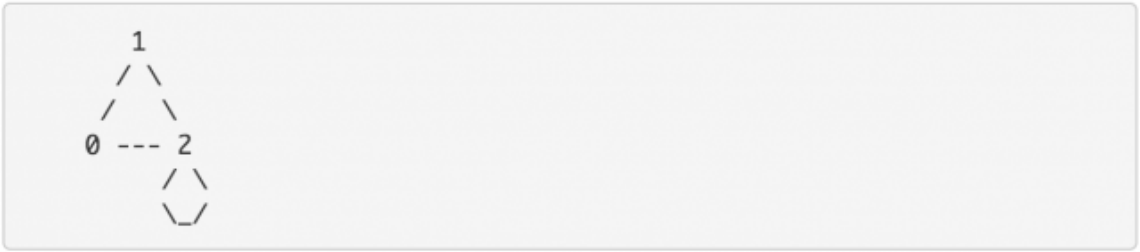
We use `#` as a separator for each node, and `,` as a separator for node label and each neighbor of the node.

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

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

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

Visually, the graph looks like the following:

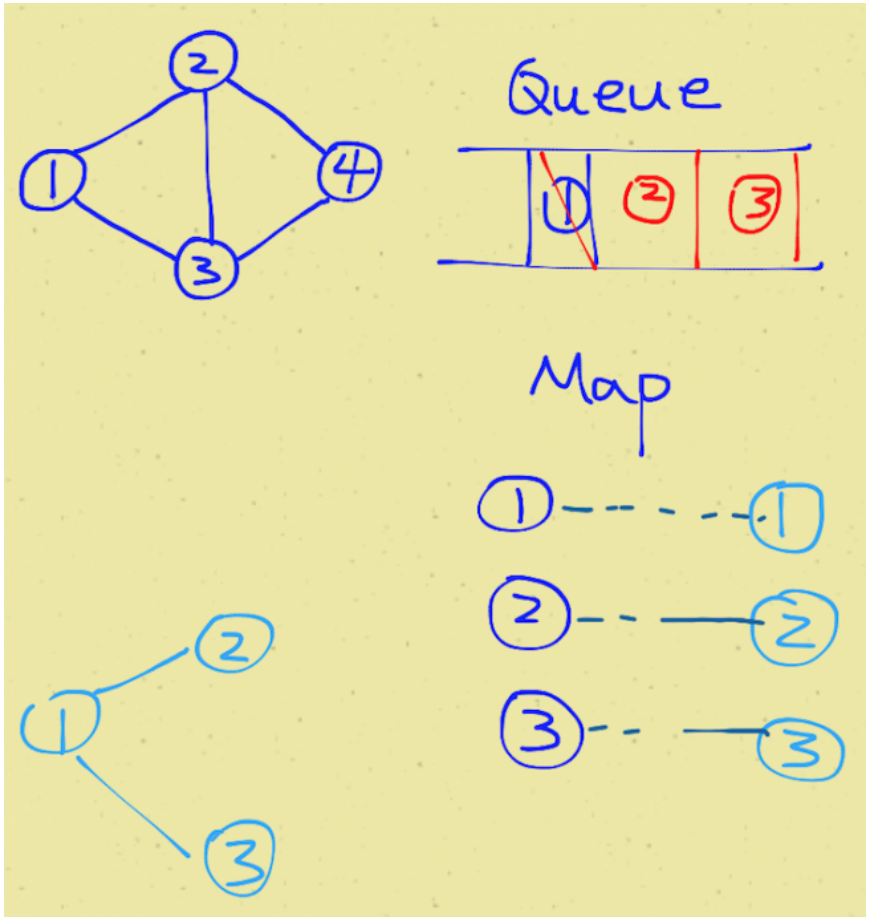



### 31.1 KEY TO SOLVE THIS PROBLEM

- A queue is used to do breath first traversal.

- a map is used to store the visited nodes. It is the map between original node and copied node.

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




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

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

```
        UndirectedGraphNode newHead = new UndirectedGraphNode(node.label);

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

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

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

        }
        return newHead;
    }
}
```

# 32

## LEETCODE SOLUTION - MERGE SORT LINKEDLIST IN JAVA

LeetCode - Sort List:

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

### 32.1 KEYS FOR SOLVING THE PROBLEM

- Break the list to two in the middle
- Recursively sort the two sub lists
- Merge the two sub lists

This is my accepted answer for the problem.

```
package algorithm.sort;

class ListNode {
        int val;
        ListNode next;

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

public class SortLinkedList {

        // merge sort
        public static ListNode mergeSortList(ListNode head) {

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

                // count total number of elements
                int count = 0;
                ListNode p = head;
                while (p != null) {
                        count++;
```

```
                p = p.next;
            }

            // break up to two list
            int middle = count / 2;

            ListNode l = head, r = null;
            ListNode p2 = head;
            int countHalf = 0;
            while (p2 != null) {
                countHalf++;
                ListNode next = p2.next;

                if (countHalf == middle) {
                    p2.next = null;
                    r = next;
                }
                p2 = next;
            }

            // now we have two parts l and r, recursively sort them
            ListNode h1 = mergeSortList(l);
            ListNode h2 = mergeSortList(r);

            // merge together
            ListNode merged = merge(h1, h2);

            return merged;
        }

        public static ListNode merge(ListNode l, ListNode r) {
            ListNode p1 = l;
            ListNode p2 = r;

            ListNode fakeHead = new ListNode(100);
            ListNode pNew = fakeHead;

            while (p1 != null || p2 != null) {

                if (p1 == null) {
                    pNew.next = new ListNode(p2.val);
                    p2 = p2.next;
                    pNew = pNew.next;
                } else if (p2 == null) {
                    pNew.next = new ListNode(p1.val);
                    p1 = p1.next;
                    pNew = pNew.next;
                } else {
                    if (p1.val < p2.val) {
                        // if(fakeHead)
                        pNew.next = new ListNode(p1.val);
                        p1 = p1.next;
```

```
                                        pNew = pNew.next;
                                } else if (p1.val == p2.val) {
                                        pNew.next = new ListNode(p1.val);
                                        pNew.next.next = new ListNode(p1.val);
                                        pNew = pNew.next.next;
                                        p1 = p1.next;
                                        p2 = p2.next;

                                } else {
                                        pNew.next = new ListNode(p2.val);
                                        p2 = p2.next;
                                        pNew = pNew.next;
                                }
                        }
                }

                // printList(fakeHead.next);
                return fakeHead.next;
        }

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

                ListNode n4 = new ListNode(3);
                ListNode n5 = new ListNode(4);
                ListNode n6 = new ListNode(5);

                n1.next = n2;
                n2.next = n3;
                n3.next = n4;
                n4.next = n5;
                n5.next = n6;

                n1 = mergeSortList(n1);

                printList(n1);
        }

        public static void printList(ListNode x) {
                if(x != null){
                        System.out.print(x.val + "␣");
                        while (x.next != null) {
                                System.out.print(x.next.val + "␣");
                                x = x.next;
                        }
                        System.out.println();
                }

        }
}
```

Output:

2 3 3 4 4 5

# 33

## QUICKSORT ARRAY IN JAVA

Quicksort is a divide and conquer algorithm. It first divides a large list into two smaller sub-lists and then recursively sort the two sub-lists. If we want to sort an array without any extra space, Quicksort is a good option. On average, time complexity is O(n log(n)).

The basic step of sorting an array are as follows:

- Select a pivot, normally the middle one
- From both ends, swap elements and make all elements on the left less than the pivot and all elements on the right greater than the pivot
- Recursively sort left part and right part

```
package algorithm.sort;

public class QuickSort {

        public static void main(String[] args) {
                int[] x = { 9, 2, 4, 7, 3, 7, 10 };
                printArray(x);

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

                quickSort(x, low, high);
                printArray(x);
        }

        public static void quickSort(int[] arr, int low, int high) {

                if (arr == null || arr.length == 0)
                        return;

                if (low >= high)
                        return;

                //pick the pivot
                int middle = low + (high - low) / 2;
                int pivot = arr[middle];

                //make left < pivot and right > pivot
```

```
                int i = low, j = high;
                while (i <= j) {
                        while (arr[i] < pivot) {
                                i++;
                        }

                        while (arr[j] > pivot) {
                                j--;
                        }

                        if (i <= j) {
                                int temp = arr[i];
                                arr[i] = arr[j];
                                arr[j] = temp;
                                i++;
                                j--;
                        }
                }

                //recursively sort two sub parts
                if (low < j)
                        quickSort(arr, low, j);

                if (high > i)
                        quickSort(arr, i, high);
        }

        public static void printArray(int[] x) {
                for (int a : x)
                        System.out.print(a + "␣");
                System.out.println();
        }
}
```

Output:

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

The mistake I made is selecting the middle element. The middle element is not (low+high)/2, but low + (high-low)/2. For other parts of the programs, just follow the algorithm.

# 34

## LEETCODE SOLUTION - SORT A LINKED LIST USING INSERTION SORT IN JAVA

Insertion Sort List:

> *Sort a linked list using insertion sort.*

This is my accepted answer for LeetCode problem - Sort a linked list using insertion sort in Java. It is a complete program.

Before coding for that, here is an example of insertion sort from wiki. You can get an idea of what is insertion sort.

3 7 4 9 5 2 6 1

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

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

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

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

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

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

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

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

Code:

```
package algorithm.sort;

class ListNode {
        int val;
        ListNode next;

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

```
}

public class SortLinkedList {
        public static ListNode insertionSortList(ListNode head) {

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

                ListNode newHead = new ListNode(head.val);
                ListNode pointer = head.next;

                // loop through each element in the list
                while (pointer != null) {
                        // insert this element to the new list

                        ListNode innerPointer = newHead;
                        ListNode next = pointer.next;

                        if (pointer.val <= newHead.val) {
                                ListNode oldHead = newHead;
                                newHead = pointer;
                                newHead.next = oldHead;
                        } else {
                                while (innerPointer.next != null) {

                                        if (pointer.val > innerPointer.val &&
                                            pointer.val <= innerPointer.next.val)
                                             {
                                                ListNode oldNext = innerPointer.
                                                    next;
                                                innerPointer.next = pointer;
                                                pointer.next = oldNext;
                                        }

                                        innerPointer = innerPointer.next;
                                }

                                if (innerPointer.next == null && pointer.val >
                                    innerPointer.val) {
                                        innerPointer.next = pointer;
                                        pointer.next = null;
                                }
                        }

                        // finally
                        pointer = next;
                }

                return newHead;
        }

        public static void main(String[] args) {
```

```
                ListNode n1 = new ListNode(2);
                ListNode n2 = new ListNode(3);
                ListNode n3 = new ListNode(4);

                ListNode n4 = new ListNode(3);
                ListNode n5 = new ListNode(4);
                ListNode n6 = new ListNode(5);

                n1.next = n2;
                n2.next = n3;
                n3.next = n4;
                n4.next = n5;
                n5.next = n6;

                n1 = insertionSortList(n1);

                printList(n1);

        }

        public static void printList(ListNode x) {
                if(x != null){
                        System.out.print(x.val + "␣");
                        while (x.next != null) {
                                System.out.print(x.next.val + "␣");
                                x = x.next;
                        }
                        System.out.println();
                }

        }
}
```

Output:

2 3 3 4 4 5

# 35

# ITERATION VS. RECURSION IN JAVA

## 35.1 RECURSION

Consider the factorial function: n!=n*(n-1)*(n-2)*...*1

There are many ways to compute factorials. One way is that n! is equal to n*(n-1)!. Therefore the program can be directly written as:

Program 1:

```
int factorial (int n) {
    if (n == 1) {
        return 1;
    } else {
        return n*factorial(n-1);
    }
}
```

In order to run this program, the computer needs to build up a chain of multiplications: factorial(n) → factorial(n-1) → factorial(n-2) → âĂę → factorial(1). Therefore, the computer has to keep track of the multiplications to be performed later on. This type of program, characterized by a chain of operations, is called recursion. Recursion can be further categorized into linear and tree recursion. When the amount of information needed to keep track of the chain of operations grows linearly with the input, the recursion is called linear recursion. The computation of n! is such a case, because the time required grows linearly with n. Another type of recursion, tree recursion, happens when the amount of information grows exponentially with the input. But we will leave it undiscussed here and go back shortly afterwards.

## 35.2 ITERATION

A different perspective on computing factorials is by first multiplying 1 by 2, then multiplying the result by 3, then by 4, and so on until n. More formally, the program can use a counter that counts from 1 up to n and compute the product simultaneously until the counter exceeds n. Therefore the program can be written as:

Program 2:

```
int factorial (int n) {
```

```
    int product = 1;
    for(int i=2; i<n; i++) {
        product *= i;
    }
    return product;
}
```

This program, by contrast to program 2, does not build a chain of multiplication. At each step, the computer only need to keep track of the current values of the product and i. This type of program is called iteration, whose state can be summarized by a fixed number of variables, a fixed rule that describes how the variables should be updated, and an end test that specifies conditions under which the process should terminate. Same as recursion, when the time required grows linearly with the input, we call the iteration linear recursion.

## 35.3 RECURSION VS ITERATION

Compared the two processes, we can find that they seem almost same, especially in term of mathematical function. They both require a number of steps proportional to n to compute n!. On the other hand, when we consider the running processes of the two programs, they evolve quite differently.

In the iterative case, the program variables provide a complete description of the state. If we stopped the computation in the middle, to resume it only need to supply the computer with all variables. However, in the recursive process, information is maintained by the computer, therefore "hidden" to the program. This makes it almost impossible to resume the program after stopping it.

## 35.4 TREE RECURSION

As described above, tree recursion happens when the amount of information grows exponentially with the input. For instance, consider the sequence of Fibonacci numbers defined as follows:

$$\mathrm{Fib}(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & \text{otherwise} \end{cases}$$

By the definition, Fibonacci numbers have the following sequence, where each number is the sum of the previous two: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

A recursive program can be immediately written as:

Program 3:

```
int fib (int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
```

```
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

Therefore, to compute fib(5), the program computes fib(4) and fib(3). To computer fib(4), it computes fib(3) and fib(2). Notice that the fib procedure calls itself twice at the last line. Two observations can be obtained from the definition and the program:

- The ith Fibonacci number Fib(i) is equal to Îęi/âĹŽ5 rounded to the nearest integer, which indicates that Fibonacci numbers grow exponentially.
- This is a bad way to compute Fibonacci numbers because it does redundant computation. Computing the running time of this procedure is beyond the scope of this article, but one can easily find that in books of algorithms, which is O(Îęn). Thus, the program takes an amount of time that grows exponentially with the input.

On the other hand, we can also write the program in an iterative way for computing the Fibonacci numbers. Program 4 is a linear iteration. The difference in time required by Program 3 and 4 is enormous, even for small inputs.

Program 4:

```
int fib (int n) {
    int fib = 0;
    int a = 1;
    for(int i=0; i<n; i++) {
        fib = fib + a;
        a = fib;
    }
    return fib;
}
```

However, one should not think tree-recursive programs are useless. When we consider programs that operate on hierarchically data structures rather than numbers, tree-recursion is a natural and powerful tool. It can help us understand and design programs. Compared with Program 3 and 4, we can easily tell Program 3 is more straightforward, even if less efficient. After that, we can most likely reformulate the program into an iterative way.

# 36

## EDIT DISTANCE IN JAVA

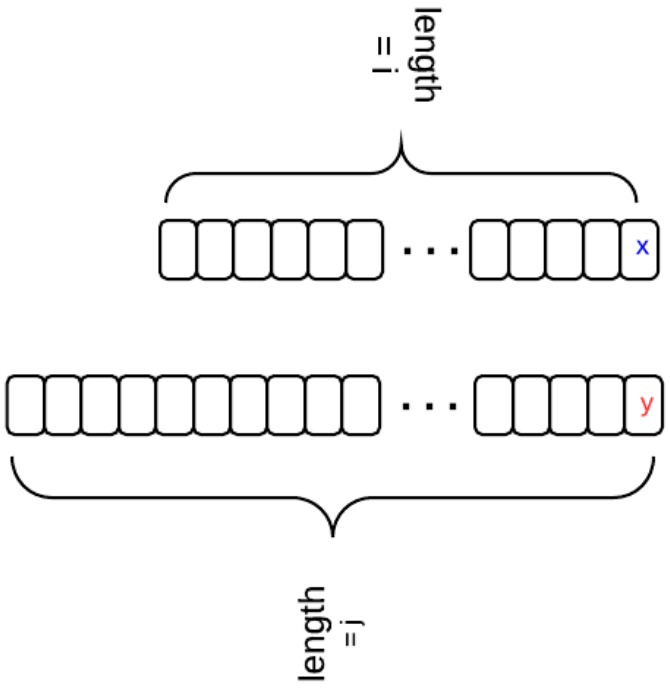
From Wiki:

> *In computer science, edit distance is a way of quantifying how dissimilar two strings (e.g., words) are to one another by counting the minimum number of operations required to transform one string into the other.*

There are three operations permitted on a word: replace, delete, insert. For example, the edit distance between "a" and "b" is 1, the edit distance between "abc" and "def" is 3. This post analyzes how to calculate edit distance by using dynamic programming.

### 36.1 KEY ANALYSIS

Let dp[i][j] stands for the edit distance between two strings with length i and j, i.e., word1[0,...,i-1] and word2[0,...,j-1]. There is a relation between dp[i][j] and dp[i-1][j-1]. Let's say we transform from one string to another. The first string has length i and it's last character is "x"; the second string has length j and its last character is "y". The following diagram shows the relation.




- if x == y, then dp[i][j] == dp[i-1][j-1]
- if x != y, and we insert y for word1, then dp[i][j] = dp[i][j-1] + 1
- if x != y, and we delete x for word1, then dp[i][j] = dp[i-1][j] + 1
- if x != y, and we replace x with y for word1, then dp[i][j] = dp[i-1][j-1] + 1
- When x!=y, dp[i][j] is the min of the three situations.

Initial condition: dp[i][0] = i, dp[0][j] = j

## 36.2 JAVA CODE

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

```
public static int minDistance(String word1, String word2) {
        int len1 = word1.length();
        int len2 = word2.length();

        // len1+1, len2+1, because finally return dp[len1][len2]
        int[][] dp = new int[len1 + 1][len2 + 1];

        for (int i = 0; i <= len1; i++) {
                dp[i][0] = i;
        }

        for (int j = 0; j <= len2; j++) {
                dp[0][j] = j;
```

```
        }

        //iterate though, and check last char
        for (int i = 0; i < len1; i++) {
                char c1 = word1.charAt(i);
                for (int j = 0; j < len2; j++) {
                        char c2 = word2.charAt(j);

                        //if last two chars equal
                        if (c1 == c2) {
                                //update dp value for +1 length
                                dp[i + 1][j + 1] = dp[i][j];
                        } else {
                                int replace = dp[i][j] + 1;
                                int insert = dp[i][j + 1] + 1;
                                int delete = dp[i + 1][j] + 1;

                                int min = replace > insert ? insert : replace;
                                min = delete > min ? min : delete;
                                dp[i + 1][j + 1] = min;
                        }
                }
        }

        return dp[len1][len2];
}
```

# 37

## LEETCODE SOLUTION OF LONGEST PALINDROMIC SUBSTRING IN JAVA

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

### 37.1 NAIVE APPROACH

Naively, we can simply examine every substring and check if it is palindromic. The time complexity is O(n3̂). If this is submitted to LeetCode onlinejudge, an error message will be returned - "Time Limit Exceeded". Therefore, this approach is just a start, we need better algorithm.

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

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

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

        return longestPalindrome;
}

public static boolean isPalindrome(String s) {

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

```
                }
        }

        return true;
}
```

## 37.2 DYNAMIC PROGRAMMING

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

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

Start condition:

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

Changing condition:

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

Time O(n2) Space O(n2)

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

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

        int maxLen = 0;
        String longestStr = null;

        int length = s.length();

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

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

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

```
        printTable(table);
        //condition for calculate whole table
        for (int l = 3; l <= length; l++) {
                for (int i = 0; i <= length-l; i++) {
                        int j = i + l - 1;
                        if (s.charAt(i) == s.charAt(j)) {
                                table[i][j] = table[i + 1][j - 1];
                                if (table[i][j] == 1 && l > maxLen)
                                        longestStr = s.substring(i, j + 1);
                        } else {
                                table[i][j] = 0;
                        }
                        printTable(table);
                }
        }

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

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

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

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

## 37.3 SIMPLE ALGORITHM

Time O(n2̂), Space O(1)

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

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

```
        }

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

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

        return longest;
}

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

## 37.4 MANACHER'S ALGORITHM

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

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

# 38

## LEETCODE SOLUTION - WORD BREAK

This is my solution for Word Break in Java.

### 38.1 THE PROBLEM

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

### 38.2 NAIVE APPROACH

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

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

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

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

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

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

```
        return false;
    }
}
```

Time: O(2n̂)

## 38.3 DYNAMIC PROGRAMMING

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

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

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

        for(int i=0; i<s.length(); i++){
            //should continue from match position
            if(!t[i])
                continue;

            for(String a: dict){
                int len = a.length();
                int end = i + len;
                if(end > s.length())
                    continue;
                if(s.substring(i, end).equals(a)){
                    t[end] = true;
                }
            }
        }

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

Time: O(string length * dict size)