

# **Coding Interview in Java**

**Program Creek**

May 1st, 2016

# Contents

<b>1</b>	<b>Rotate Array in Java</b>	<b>13</b>
<b>2</b>	<b>Reverse Words in a String II</b>	<b>17</b>
<b>3</b>	<b>Evaluate Reverse Polish Notation</b>	<b>19</b>
<b>4</b>	<b>Isomorphic Strings</b>	<b>23</b>
<b>5</b>	<b>Word Ladder</b>	<b>25</b>
<b>6</b>	<b>Word Ladder II</b>	<b>27</b>
<b>7</b>	<b>Median of Two Sorted Arrays</b>	<b>31</b>
<b>8</b>	<b>Kth Largest Element in an Array</b>	<b>33</b>
<b>9</b>	<b>Wildcard Matching</b>	<b>35</b>
<b>10</b>	<b>Regular Expression Matching in Java</b>	<b>37</b>
<b>11</b>	<b>Merge Intervals</b>	<b>41</b>
<b>12</b>	<b>Insert Interval</b>	<b>43</b>
<b>13</b>	<b>Two Sum</b>	<b>45</b>
<b>14</b>	<b>Two Sum II Input array is sorted</b>	<b>47</b>
<b>15</b>	<b>Two Sum III Data structure design</b>	<b>49</b>
<b>16</b>	<b>3Sum</b>	<b>51</b>
<b>17</b>	<b>4Sum</b>	<b>55</b>
<b>18</b>	<b>3Sum Closest</b>	<b>57</b>
<b>19</b>	<b>String to Integer (atoi)</b>	<b>59</b>
<b>20</b>	<b>Merge Sorted Array</b>	<b>61</b>

<b>21</b>	<b>Valid Parentheses</b>	<b>63</b>
<b>22</b>	<b>Longest Valid Parentheses</b>	<b>65</b>
<b>23</b>	<b>Implement strStr()</b>	<b>67</b>
<b>24</b>	<b>Minimum Size Subarray Sum</b>	<b>71</b>
<b>25</b>	<b>Search Insert Position</b>	<b>75</b>
<b>26</b>	<b>Longest Consecutive Sequence</b>	<b>77</b>
<b>27</b>	<b>Valid Palindrome</b>	<b>79</b>
<b>28</b>	<b>ZigZag Conversion</b>	<b>83</b>
<b>29</b>	<b>Add Binary</b>	<b>85</b>
<b>30</b>	<b>Length of Last Word</b>	<b>87</b>
<b>31</b>	<b>Triangle</b>	<b>89</b>
<b>32</b>	<b>Contains Duplicate</b>	<b>91</b>
<b>33</b>	<b>Contains Duplicate II</b>	<b>93</b>
<b>34</b>	<b>Contains Duplicate III</b>	<b>95</b>
<b>35</b>	<b>Remove Duplicates from Sorted Array</b>	<b>97</b>
<b>36</b>	<b>Remove Duplicates from Sorted Array II</b>	<b>101</b>
<b>37</b>	<b>Remove Element</b>	<b>103</b>
<b>38</b>	<b>Move Zeroes</b>	<b>105</b>
<b>39</b>	<b>Longest Substring Without Repeating Characters</b>	<b>107</b>
<b>40</b>	<b>Longest Substring Which Contains 2 Unique Characters</b>	<b>111</b>
<b>41</b>	<b>Substring with Concatenation of All Words</b>	<b>115</b>
<b>42</b>	<b>Minimum Window Substring</b>	<b>117</b>
<b>43</b>	<b>Reverse Words in a String</b>	<b>119</b>
<b>44</b>	<b>Find Minimum in Rotated Sorted Array</b>	<b>121</b>
<b>45</b>	<b>Find Minimum in Rotated Sorted Array II</b>	<b>123</b>

## Contents

---

<b>46 Search in Rotated Sorted Array</b>	<b>125</b>
<b>47 Search in Rotated Sorted Array II</b>	<b>127</b>
<b>48 Min Stack</b>	<b>129</b>
<b>49 Majority Element</b>	<b>131</b>
<b>50 Majority Element II</b>	<b>133</b>
<b>51 Bulls and Cows</b>	<b>135</b>
<b>52 Largest Rectangle in Histogram</b>	<b>139</b>
<b>53 Longest Common Prefix</b>	<b>141</b>
<b>54 Largest Number</b>	<b>143</b>
<b>55 Simplify Path</b>	<b>145</b>
<b>56 Compare Version Numbers</b>	<b>147</b>
<b>57 Gas Station</b>	<b>149</b>
<b>58 Pascal's Triangle</b>	<b>151</b>
<b>59 Pascal's Triangle II</b>	<b>153</b>
<b>60 Container With Most Water</b>	<b>155</b>
<b>61 Candy</b>	<b>157</b>
<b>62 Trapping Rain Water</b>	<b>159</b>
<b>63 Count and Say</b>	<b>161</b>
<b>64 Search for a Range</b>	<b>163</b>
<b>65 Basic Calculator</b>	<b>165</b>
<b>66 Group Anagrams</b>	<b>167</b>
<b>67 Shortest Palindrome</b>	<b>169</b>
<b>68 Rectangle Area</b>	<b>171</b>
<b>69 Summary Ranges</b>	<b>173</b>
<b>70 Increasing Triplet Subsequence</b>	<b>175</b>

<b>71</b>	<b>Get Target Number Using Number List and Arithmetic Operations</b>	<b>177</b>
<b>72</b>	<b>Reverse Vowels of a String</b>	<b>179</b>
<b>73</b>	<b>Flip Game</b>	<b>181</b>
<b>74</b>	<b>Flip Game II</b>	<b>183</b>
<b>75</b>	<b>Missing Number</b>	<b>185</b>
<b>76</b>	<b>Valid Anagram</b>	<b>187</b>
<b>77</b>	<b>Group Shifted Strings</b>	<b>189</b>
<b>78</b>	<b>Top K Frequent Elements</b>	<b>191</b>
<b>79</b>	<b>Find Peak Element</b>	<b>195</b>
<b>80</b>	<b>Word Pattern</b>	<b>197</b>
<b>81</b>	<b>HIndex</b>	<b>199</b>
<b>82</b>	<b>Palindrome Pairs</b>	<b>201</b>
<b>83</b>	<b>One Edit Distance</b>	<b>205</b>
<b>84</b>	<b>Scramble String</b>	<b>207</b>
<b>85</b>	<b>First Bad Version</b>	<b>209</b>
<b>86</b>	<b>Set Matrix Zeroes</b>	<b>211</b>
<b>87</b>	<b>Spiral Matrix</b>	<b>215</b>
<b>88</b>	<b>Spiral Matrix II</b>	<b>219</b>
<b>89</b>	<b>Search a 2D Matrix</b>	<b>221</b>
<b>90</b>	<b>Search a 2D Matrix II</b>	<b>223</b>
<b>91</b>	<b>Rotate Image</b>	<b>227</b>
<b>92</b>	<b>Valid Sudoku</b>	<b>229</b>
<b>93</b>	<b>Minimum Path Sum</b>	<b>231</b>
<b>94</b>	<b>Unique Paths</b>	<b>233</b>
<b>95</b>	<b>Unique Paths II</b>	<b>235</b>

## Contents

---

<b>96 Number of Islands</b>	<b>237</b>
<b>97 Number of Islands II</b>	<b>239</b>
<b>98 Surrounded Regions</b>	<b>241</b>
<b>99 Maximal Rectangle</b>	<b>245</b>
<b>100 Maximal Square</b>	<b>247</b>
<b>101 Word Search</b>	<b>249</b>
<b>102 Word Search II</b>	<b>251</b>
<b>103 Integer Break</b>	<b>255</b>
<b>104 Range Sum Query 2D Immutable</b>	<b>257</b>
<b>105 Longest Increasing Path in a Matrix</b>	<b>259</b>
<b>106 Shortest Distance from All Buildings</b>	<b>261</b>
<b>107 Game of Life</b>	<b>265</b>
<b>108 Implement a Stack Using an Array in Java</b>	<b>267</b>
<b>109 Add Two Numbers</b>	<b>271</b>
<b>110 Reorder List</b>	<b>273</b>
<b>111 Linked List Cycle</b>	<b>279</b>
<b>112 Copy List with Random Pointer</b>	<b>281</b>
<b>113 Merge Two Sorted Lists</b>	<b>285</b>
<b>114 Odd Even Linked List</b>	<b>287</b>
<b>115 Remove Duplicates from Sorted List</b>	<b>289</b>
<b>116 Remove Duplicates from Sorted List II</b>	<b>291</b>
<b>117 Partition List</b>	<b>293</b>
<b>118 LRU Cache</b>	<b>295</b>
<b>119 Intersection of Two Linked Lists</b>	<b>299</b>
<b>120 Remove Linked List Elements</b>	<b>301</b>

<b>121 Swap Nodes in Pairs</b>	303
<b>122 Reverse Linked List</b>	305
<b>123 Reverse Linked List II</b>	307
<b>124 Remove Nth Node From End of List</b>	309
<b>125 Implement Stack using Queues</b>	311
<b>126 Implement Queue using Stacks</b>	313
<b>127 Palindrome Linked List</b>	315
<b>128 Implement a Queue using an Array in Java</b>	317
<b>129 Delete Node in a Linked List</b>	319
<b>130 Moving Average from Data Stream</b>	321
<b>131 Java PriorityQueue Class Example</b>	323
<b>132 Binary Tree Preorder Traversal</b>	325
<b>133 Binary Tree Inorder Traversal</b>	327
<b>134 Binary Tree Postorder Traversal</b>	329
<b>135 Binary Tree Level Order Traversal</b>	335
<b>136 Binary Tree Level Order Traversal II</b>	337
<b>137 Binary Tree Vertical Order Traversal</b>	339
<b>138 Invert Binary Tree</b>	341
<b>139 Kth Smallest Element in a BST</b>	343
<b>140 Binary Tree Longest Consecutive Sequence</b>	345
<b>141 Validate Binary Search Tree</b>	349
<b>142 Flatten Binary Tree to Linked List</b>	351
<b>143 Path Sum</b>	353
<b>144 Path Sum II</b>	355
<b>145 Construct Binary Tree from Inorder and Postorder Traversal</b>	357

## Contents

---

<b>146 Construct Binary Tree from Preorder and Inorder Traversal</b>	359
<b>147 Convert Sorted Array to Binary Search Tree</b>	361
<b>148 Convert Sorted List to Binary Search Tree</b>	363
<b>149 Minimum Depth of Binary Tree</b>	365
<b>150 Binary Tree Maximum Path Sum</b>	367
<b>151 Balanced Binary Tree</b>	369
<b>152 Symmetric Tree</b>	371
<b>153 Binary Search Tree Iterator</b>	373
<b>154 Binary Tree Right Side View</b>	375
<b>155 Lowest Common Ancestor of a Binary Search Tree</b>	377
<b>156 Lowest Common Ancestor of a Binary Tree</b>	379
<b>157 Verify Preorder Serialization of a Binary Tree</b>	381
<b>158 Populating Next Right Pointers in Each Node</b>	383
<b>159 Populating Next Right Pointers in Each Node II</b>	385
<b>160 Unique Binary Search Trees</b>	387
<b>161 Unique Binary Search Trees II</b>	389
<b>162 Sum Root to Leaf Numbers</b>	391
<b>163 Count Complete Tree Nodes</b>	393
<b>164 Closest Binary Search Tree Value</b>	395
<b>165 Binary Tree Paths</b>	397
<b>166 Maximum Depth of Binary Tree</b>	399
<b>167 Recover Binary Search Tree</b>	401
<b>168 Merge K Sorted Arrays in Java</b>	403
<b>169 Merge k Sorted Lists</b>	405
<b>170 Find Median from Data Stream</b>	407

<b>171 Implement Trie (Prefix Tree)</b>	<b>409</b>
<b>172 Add and Search Word Data structure design</b>	<b>413</b>
<b>173 Range Sum Query Mutable</b>	<b>417</b>
<b>174 The Skyline Problem</b>	<b>421</b>
<b>175 Clone Graph Java</b>	<b>423</b>
<b>176 Course Schedule</b>	<b>427</b>
<b>177 Course Schedule II</b>	<b>431</b>
<b>178 Reconstruct Itinerary</b>	<b>433</b>
<b>179 Graph Valid Tree</b>	<b>435</b>
<b>180 How Developers Sort in Java?</b>	<b>439</b>
<b>181 Solution Merge Sort LinkedList in Java</b>	<b>441</b>
<b>182 Quicksort Array in Java</b>	<b>445</b>
<b>183 Solution Sort a linked list using insertion sort in Java</b>	<b>447</b>
<b>184 Maximum Gap</b>	<b>451</b>
<b>185 First Missing Positive</b>	<b>453</b>
<b>186 Sort Colors</b>	<b>455</b>
<b>187 Edit Distance in Java</b>	<b>457</b>
<b>188 Distinct Subsequences Total</b>	<b>461</b>
<b>189 Longest Palindromic Substring</b>	<b>463</b>
<b>190 Word Break</b>	<b>467</b>
<b>191 Word Break II</b>	<b>471</b>
<b>192 Maximum Subarray</b>	<b>475</b>
<b>193 Maximum Product Subarray</b>	<b>477</b>
<b>194 Palindrome Partitioning</b>	<b>479</b>
<b>195 Palindrome Partitioning II</b>	<b>483</b>

## Contents

---

<b>196 House Robber</b>	485
<b>197 House Robber II</b>	487
<b>198 House Robber III</b>	489
<b>199 Jump Game</b>	491
<b>200 Jump Game II</b>	493
<b>201 Best Time to Buy and Sell Stock</b>	495
<b>202 Best Time to Buy and Sell Stock II</b>	497
<b>203 Best Time to Buy and Sell Stock III</b>	499
<b>204 Best Time to Buy and Sell Stock IV</b>	501
<b>205 Dungeon Game</b>	503
<b>206 Decode Ways</b>	505
<b>207 Longest Common Subsequence</b>	507
<b>208 Longest Common Substring</b>	509
<b>209 Longest Increasing Subsequence</b>	511
<b>210 Coin Change</b>	515
<b>211 Perfect Squares</b>	517
<b>212 Single Number</b>	519
<b>213 Single Number II</b>	521
<b>214 Twitter Codility Problem Max Binary Gap</b>	523
<b>215 Number of 1 Bits</b>	525
<b>216 Reverse Bits</b>	527
<b>217 Repeated DNA Sequences</b>	529
<b>218 Bitwise AND of Numbers Range</b>	531
<b>219 Power of Two</b>	533
<b>220 Counting Bits</b>	535

<b>221 Maximum Product of Word Lengths</b>	537
<b>222 Permutations</b>	539
<b>223 Permutations II</b>	543
<b>224 Permutation Sequence</b>	545
<b>225 Generate Parentheses</b>	547
<b>226 Combination Sum</b>	549
<b>227 Combination Sum II</b>	551
<b>228 Combination Sum III</b>	553
<b>229 Combinations</b>	555
<b>230 Letter Combinations of a Phone Number</b>	559
<b>231 Restore IP Addresses</b>	561
<b>232 Reverse Integer</b>	563
<b>233 Palindrome Number</b>	565
<b>234 Pow(x, n)</b>	567

## Contents

---

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

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

This collection is updated from time to time. Please check out this link for the latest version: <http://www.programcreek.com/2012/11/top-10-algorithms-for-coding-interview/>

# 1 Rotate Array in Java

## 1.1 Problem

Rotate an array of n elements to the right by k steps.

For example, with n = 7 and k = 3, the array [1,2,3,4,5,6,7] is rotated to [5,6,7,1,2,3,4]. How many different ways do you know to solve this problem?

## 1.2 Solution 1 - Intermediate Array

In a straightforward way, we can create a new array and then copy elements to the new array. Then change the original array by using System.arraycopy().

```
public void rotate(int[] nums, int k) {  
    if(k > nums.length)  
        k=k%nums.length;  
  
    int[] result = new int[nums.length];  
  
    for(int i=0; i < k; i++){  
        result[i] = nums[nums.length-k+i];  
    }  
  
    int j=0;  
    for(int i=k; i<nums.length; i++){  
        result[i] = nums[j];  
        j++;  
    }  
  
    System.arraycopy( result, 0, nums, 0, nums.length );  
}
```

Space is O(n) and time is O(n). You can check out the difference between System.arraycopy() and Arrays.copyOf().

## 1.3 Solution 2 - Bubble Rotate

Can we do this in O(1) space?

This solution is like a bubble sort.

```
public static void rotate(int[] arr, int order) {  
    if (arr == null || order < 0) {
```

## 1 Rotate Array in Java

---

```
        throw new IllegalArgumentException("Illegal argument!");
    }

    for (int i = 0; i < order; i++) {
        for (int j = arr.length - 1; j > 0; j--) {
            int temp = arr[j];
            arr[j] = arr[j - 1];
            arr[j - 1] = temp;
        }
    }
}
```

---

However, the time is  $O(n*k)$ .

Here is an example (length=7, order=3):

```
i=0
0 1 2 3 4 5 6
0 1 2 3 4 6 5
...
6 0 1 2 3 4 5
i=1
6 0 1 2 3 5 4
...
5 6 0 1 2 3 4
i=2
5 6 0 1 2 4 3
...
4 5 6 0 1 2 3
```

---

## 1.4 Solution 3 - Reversal

Can we do this in  $O(1)$  space and in  $O(n)$  time? The following solution does.

Assuming we are given 1,2,3,4,5,6 and order 2. The basic idea is:

1. Divide the array two parts: 1,2,3,4 and 5, 6
  2. Reverse first part: 4,3,2,1,5,6
  3. Reverse second part: 4,3,2,1,6,5
  4. Reverse the whole array: 5,6,1,2,3,4
- 

```
public static void rotate(int[] arr, int order) {
    if (arr == null || arr.length==0 || order < 0) {
        throw new IllegalArgumentException("Illegal argument!");
    }

    if(order > arr.length){
        order = order %arr.length;
    }
```

```
//length of first part
int a = arr.length - order;

reverse(arr, 0, a-1);
reverse(arr, a, arr.length-1);
reverse(arr, 0, arr.length-1);

}

public static void reverse(int[] arr, int left, int right){
    if(arr == null || arr.length == 1)
        return;

    while(left < right){
        int temp = arr[left];
        arr[left] = arr[right];
        arr[right] = temp;
        left++;
        right--;
    }
}
```

---



## 2 Reverse Words in a String II

Given an input string, reverse the string word by word. A word is defined as a sequence of non-space characters.

The input string does not contain leading or trailing spaces and the words are always separated by a single space.

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

Could you do it in-place without allocating extra space?

### 2.1 Java Solution

---

```
public void reverseWords(char[] s) {
    int i=0;
    for(int j=0; j<s.length; j++){
        if(s[j]==' '){
            reverse(s, i, j-1);
            i=j+1;
        }
    }

    reverse(s, i, s.length-1);

    reverse(s, 0, s.length-1);
}

public void reverse(char[] s, int i, int j){
    while(i<j){
        char temp = s[i];
        s[i]=s[j];
        s[j]=temp;
        i++;
        j--;
    }
}
```

---



## 3 Evaluate Reverse Polish Notation

Evaluate the value of an arithmetic expression in Reverse Polish Notation. Valid operators are +, -, \*, /. Each operand may be an integer or another expression. For example:

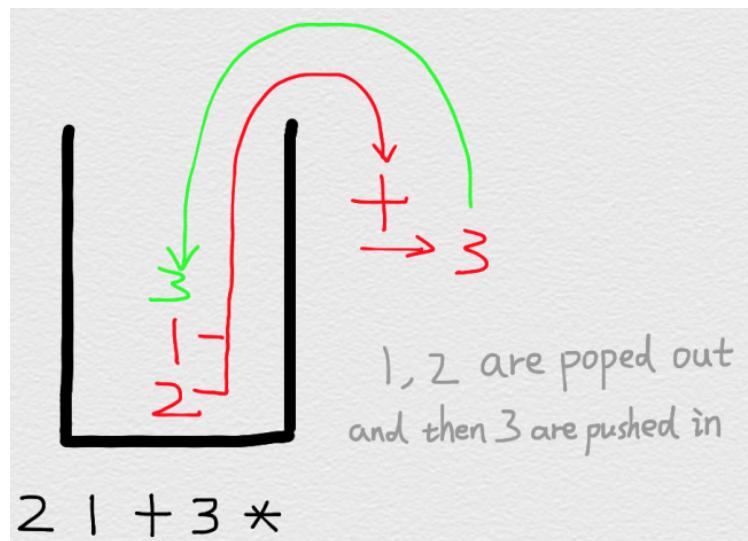
---

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

---

### 3.1 Naive Approach

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. 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));  
    }  
}
```

### 3 Evaluate Reverse Polish Notation

---

```
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)) { //push to stack if it is a number
            stack.push(t);
        } else { //pop numbers from stack if it is an operator
            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 a JDK version below 1.7.

## 3.2 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;

}
```

---



# 4 Isomorphic Strings

Given two strings s and t, determine if they are isomorphic. Two strings are isomorphic if the characters in s can be replaced to get t.

For example, "egg" and "add" are isomorphic, "foo" and "bar" are not.

## 4.1 Analysis

We can define a map which tracks the char-char mappings. If a value is already mapped, it can not be mapped again.

## 4.2 Java Solution

---

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

    if(s.length()!=t.length())
        return false;

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

    for(int i=0; i<s.length(); i++){
        char c1 = s.charAt(i);
        char c2 = t.charAt(i);

        if(map.containsKey(c1)){
            if(map.get(c1)!=c2)// if not consistant with previous ones
                return false;
        }else{
            if(map.containsValue(c2)) //if c2 is already being mapped
                return false;
            map.put(c1, c2);
        }
    }

    return true;
}
```

---

Time is O(n).



# 5 Word Ladder

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 and each intermediate word must exist in the dictionary. For example, given:

---

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

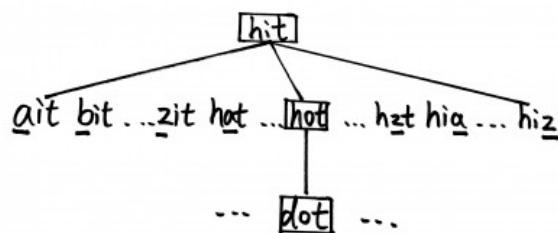
---

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

## 5.1 Analysis

UPDATED on 06/07/2015.

So we quickly realize that this is a search problem, and breath-first search guarantees the optimal solution.



## 5.2 Java Solution

---

```
class WordNode{  
    String word;  
    int numSteps;  
  
    public WordNode(String word, int numSteps){  
        this.word = word;  
        this.numSteps = numSteps;  
    }  
}
```

## 5 Word Ladder

---

```
public class Solution {
    public int ladderLength(String beginWord, String endWord, Set<String>
    wordDict) {
        LinkedList<WordNode> queue = new LinkedList<WordNode>();
        queue.add(new WordNode(beginWord, 1));

        wordDict.add(endWord);

        while(!queue.isEmpty()){
            WordNode top = queue.remove();
            String word = top.word;

            if(word.equals(endWord)){
                return top.numSteps;
            }

            char[] arr = word.toCharArray();
            for(int i=0; i<arr.length; i++){
                for(char c='a'; c<='z'; c++){
                    char temp = arr[i];
                    if(arr[i]!=c){
                        arr[i]=c;
                    }

                    String newWord = new String(arr);
                    if(wordDict.contains(newWord)){
                        queue.add(new WordNode(newWord, top.numSteps+1));
                        wordDict.remove(newWord);
                    }

                    arr[i]=temp;
                }
            }
        }

        return 0;
    }
}
```

---

# 6 Word Ladder II

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

- 1) Only one letter can be changed at a time,
- 2) Each intermediate word must exist in the dictionary.

For example, given: start = "hit", end = "cog", and dict = ["hot", "dot", "dog", "lot", "log"], return:

---

```
[  
    ["hit", "hot", "dot", "dog", "cog"],  
    ["hit", "hot", "lot", "log", "cog"]  
]
```

---

## 6.1 Analysis

This is an extension of [Word Ladder](#).

The idea is the same. To track the actual ladder, we need to add a pointer that points to the previous node in the WordNode class.

In addition, the used word can not directly removed from the dictionary. The used word is only removed when steps change.

## 6.2 Java Solution

---

```
class WordNode{  
    String word;  
    int numSteps;  
    WordNode pre;  
  
    public WordNode(String word, int numSteps, WordNode pre){  
        this.word = word;  
        this.numSteps = numSteps;  
        this.pre = pre;  
    }  
}  
  
public class Solution {  
    public List<List<String>> findLadders(String start, String end,  
        Set<String> dict) {  
        List<List<String>> result = new ArrayList<List<String>>();
```

## 6 Word Ladder II

---

```
LinkedList<WordNode> queue = new LinkedList<WordNode>();
queue.add(new WordNode(start, 1, null));

dict.add(end);

int minStep = 0;

HashSet<String> visited = new HashSet<String>();
HashSet<String> unvisited = new HashSet<String>();
unvisited.addAll(dict);

int preNumSteps = 0;

while(!queue.isEmpty()){
    WordNode top = queue.remove();
    String word = top.word;
    int currNumSteps = top.numSteps;

    if(word.equals(end)){
        if(minStep == 0){
            minStep = top.numSteps;
        }

        if(top.numSteps == minStep && minStep !=0){
            //nothing
            ArrayList<String> t = new ArrayList<String>();
            t.add(top.word);
            while(top.pre !=null){
                t.add(0, top.pre.word);
                top = top.pre;
            }
            result.add(t);
            continue;
        }
    }

    if(preNumSteps < currNumSteps){
        unvisited.removeAll(visited);
    }

    preNumSteps = currNumSteps;

    char[] arr = word.toCharArray();
    for(int i=0; i<arr.length; i++){
        for(char c='a'; c<='z'; c++){
            char temp = arr[i];
            if(arr[i]!=c){
                arr[i]=c;
            }
        }
    }
}
```

```
String newWord = new String(arr);
if(unvisited.contains(newWord)){
    queue.add(new WordNode(newWord, top.numSteps+1, top));
    visited.add(newWord);
}

arr[i]=temp;
}
}

return result;
}
}
```

---



# 7 Median of Two Sorted Arrays

*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)).*

## 7.1 Java Solution 1

If we see  $\log(n)$ , we should think about using binary something.

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 to get  $\log(n)$  time.

```
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);
}
```

---

## 7.2 Java Solution 2

Solution 1 is a general solution to find the kth element. We can also come up with a simpler solution which only finds the median of two sorted arrays for this particular problem. Thanks to Gunner86. The description of the algorithm is awesome!

---

- 1) Calculate the medians m1 and m2 of the input arrays ar1[] and ar2[] respectively.
  - 2) If m1 and m2 both are equal then we are done, and `return m1 (or m2)`
  - 3) If m1 is greater than m2, then median is present in one of the below two subarrays.
    - a) From first element of ar1 to m1 (`ar1[0...|_n/2_|]`)
    - b) From m2 to last element of ar2 (`ar2[|_n/2_|...n-1]`)
  - 4) If m2 is greater than m1, then median is present in one of the below two subarrays.
    - a) From m1 to last element of ar1 (`ar1[|_n/2_|...n-1]`)
    - b) From first element of ar2 to m2 (`ar2[0...|_n/2_|]`)
  - 5) Repeat the above process until size of both the subarrays becomes 2.
  - 6) If size of the two arrays is 2 then use below formula to get the median.  
$$\text{Median} = (\max(\text{ar1}[0], \text{ar2}[0]) + \min(\text{ar1}[1], \text{ar2}[1]))/2$$
-

# 8 Kth Largest Element in an Array

Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

For example, given [3,2,1,5,6,4] and k = 2, return 5.

Note: You may assume k is always valid,  $1 \leq k \leq$  array's length.

## 8.1 Java Solution 1 - Sorting

---

```
public int findKthLargest(int[] nums, int k) {
    Arrays.sort(nums);
    return nums[nums.length-k];
}
```

---

Time is  $O(n\log(n))$

## 8.2 Java Solution 2 - Quick Sort

This problem can also be solved by using the quickselect approach, which is similar to quicksort.

---

```
public int findKthLargest(int[] nums, int k) {
    if (k < 1 || nums == null) {
        return 0;
    }

    return getKth(nums.length - k + 1, nums, 0, nums.length - 1);
}

public int getKth(int k, int[] nums, int start, int end) {
    int pivot = nums[end];

    int left = start;
    int right = end;

    while (true) {

        while (nums[left] < pivot && left < right) {
            left++;
        }
    }
}
```

---

```
    while (nums[right] >= pivot && right > left) {
        right--;
    }

    if (left == right) {
        break;
    }

    swap(nums, left, right);
}

swap(nums, left, end);

if (k == left + 1) {
    return pivot;
} else if (k < left + 1) {
    return getKth(k, nums, start, left - 1);
} else {
    return getKth(k, nums, left + 1, end);
}
}

public void swap(int[] nums, int n1, int n2) {
    int tmp = nums[n1];
    nums[n1] = nums[n2];
    nums[n2] = tmp;
}
```

---

Average case time is  $O(n)$ , worst case time is  $O(n^2)$ .

### 8.3 Java Solution 3 - Heap

We can use a min heap to solve this problem. The heap stores the top k elements. Whenever the size is greater than k, delete the min. Time complexity is  $O(n \log(k))$ . Space complexity is  $O(k)$  for storing the top k numbers.

```
public int findKthLargest(int[] nums, int k) {
    PriorityQueue<Integer> q = new PriorityQueue<Integer>(k);
    for(int i: nums){
        q.offer(i);

        if(q.size()>k){
            q.poll();
        }
    }

    return q.peek();
}
```

---

# 9 Wildcard Matching

Implement wildcard pattern matching with support for '?' and '\*'.

## 9.1 Java Solution

To understand this solution, you can use s="aab" and p="\*ab".

---

```
public boolean isMatch(String s, String p) {
    int i = 0;
    int j = 0;
    int starIndex = -1;
    int iIndex = -1;

    while (i < s.length()) {
        if (j < p.length() && (p.charAt(j) == '?' || p.charAt(j) == s.charAt(i))) {
            {
                ++i;
                ++j;
            } else if (j < p.length() && p.charAt(j) == '*') {
                starIndex = j;
                iIndex = i;
                j++;
            } else if (starIndex != -1) {
                j = starIndex + 1;
                i = iIndex+1;
                iIndex++;
            } else {
                return false;
            }
        }

        while (j < p.length() && p.charAt(j) == '*') {
            ++j;
        }

        return j == p.length();
    }
}
```

---



# 10 Regular Expression Matching in Java

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

## 10.1 Analysis

First of all, this is one of the most difficulty problems. It is hard to think through all different cases. The problem should be simplified to handle 2 basic cases:

- the second char of pattern is "\*"
- the second char of pattern is not "\*"

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

## 10.2 Java Solution 1 (Short)

The following Java solution is accepted.

---

```
public class Solution {
    public boolean isMatch(String s, String p) {
        if(p.length() == 0)
            return s.length() == 0;

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

```
        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;
    }
}
```

---

### 10.3 Java Solution 2 (More Readable)

```
public boolean isMatch(String s, String p) {
    // base case
    if (p.length() == 0) {
        return s.length() == 0;
    }

    // special case
    if (p.length() == 1) {

        // if the length of s is 0, return false
        if (s.length() < 1) {
            return false;
        }

        //if the first does not match, return false
        else if ((p.charAt(0) != s.charAt(0)) && (p.charAt(0) != '.')) {
            return false;
        }

        // otherwise, compare the rest of the string of s and p.
        else {
            return isMatch(s.substring(1), p.substring(1));
        }
    }

    // case 1: when the second char of p is not '*'
    if (p.charAt(1) != '*') {
        if (s.length() < 1) {
            return false;
        }
        if ((p.charAt(0) != s.charAt(0)) && (p.charAt(0) != '.')) {
```

```
        return false;
    } else {
        return isMatch(s.substring(1), p.substring(1));
    }
}

// case 2: when the second char of p is '*', complex case.
else {
    //case 2.1: a char & '*' can stand for 0 element
    if (isMatch(s, p.substring(2))) {
        return true;
    }

    //case 2.2: a char & '*' can stand for 1 or more preceding element,
    //so try every sub string
    int i = 0;
    while (i<s.length() && (s.charAt(i)==p.charAt(0) || p.charAt(0)=='.')){
        if (isMatch(s.substring(i + 1), p.substring(2))) {
            return true;
        }
        i++;
    }
    return false;
}
}
```

---



# 11 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].

---

## 11.1 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.

## 11.2 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
```

## 11 Merge Intervals

---

```
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;  
    }  
}
```

---

# 12 Insert Interval

Problem:

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

Example 1:

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

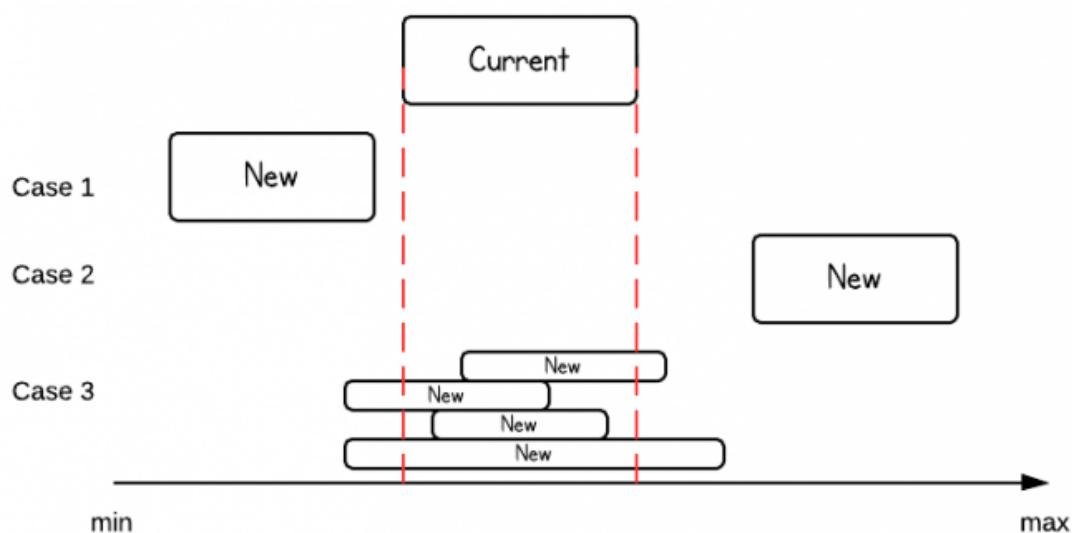
Example 2:

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

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

## 12.1 Thoughts of This Problem

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



## 12.2 Java Solution

## 12 Insert Interval

---

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

---

# 13 Two Sum

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

The function twoSum should return indices of the two numbers such that they add up to the target, where index1 must be less than index2. Please note that your returned answers (both index1 and index2) are not zero-based.

For example:

---

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

---

## 13.1 Naive Approach

This problem is pretty straightforward. We can simply examine every possible pair of numbers in this integer array.

Time complexity in worst case:  $O(n^2)$ .

---

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

---

Can we do better?

## 13.2 Better Solution

Use HashMap to store the target value.

---

```
public class Solution {  
    public int[] twoSum(int[] numbers, int target) {  
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();  
        int[] result = new int[2];
```

## 13 Two Sum

---

```
for (int i = 0; i < numbers.length; i++) {  
    if (map.containsKey(numbers[i])) {  
        int index = map.get(numbers[i]);  
        result[0] = index+1 ;  
        result[1] = i+1;  
        break;  
    } else {  
        map.put(target - numbers[i], i);  
    }  
  
}  
  
return result;  
}  
}
```

---

Time complexity depends on the put and get operations of HashMap which is normally  $O(1)$ .

Time complexity of this solution is  $O(n)$ .

## 14 Two Sum II Input array is sorted

This problem is similar to [Two Sum](#).

To solve this problem, we can use two points to scan the array from both sides. See Java solution below:

---

```
public int[] twoSum(int[] numbers, int target) {
    if (numbers == null || numbers.length == 0)
        return null;

    int i = 0;
    int j = numbers.length - 1;

    while (i < j) {
        int x = numbers[i] + numbers[j];
        if (x < target) {
            ++i;
        } else if (x > target) {
            j--;
        } else {
            return new int[] { i + 1, j + 1 };
        }
    }

    return null;
}
```

---



# 15 Two Sum III Data structure design

Design and implement a TwoSum class. It should support the following operations: add and find.

add - Add the number to an internal data structure. find - Find if there exists any pair of numbers which sum is equal to the value.

For example,

---

```
add(1);
add(3);
add(5);
find(4) -> true
find(7) -> false
```

---

## 15.1 Java Solution

Since the desired class need add and get operations, HashMap is a good option for this purpose.

```
public class TwoSum {
    private HashMap<Integer, Integer> elements = new HashMap<Integer,
        Integer>();

    public void add(int number) {
        if (elements.containsKey(number)) {
            elements.put(number, elements.get(number) + 1);
        } else {
            elements.put(number, 1);
        }
    }

    public boolean find(int value) {
        for (Integer i : elements.keySet()) {
            int target = value - i;
            if (elements.containsKey(target)) {
                if (i == target && elements.get(target) < 2) {
                    continue;
                }
                return true;
            }
        }
        return false;
    }
}
```

## 15 Two Sum III Data structure design

---

```
    }  
}
```

---

# 16 3Sum

Problem:

Given an array  $S$  of  $n$  integers, are there elements  $a, b, c$  in  $S$  such that  $a + b + c = 0$ ?

Find all unique triplets in the array which gives the sum of zero.

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

---

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

A solution set is:

$(-1, 0, 1)$   
 $(-1, -1, 2)$

---

## 16.1 Naive Solution

Naive solution is 3 loops, and this gives time complexity  $O(n^3)$ . 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

---

## 16.2 Better Solution

A better solution is using two pointers instead of one. This makes time complexity of  $O(n^2)$ .

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;
}
```

---



## 17 4Sum

Given an array S of n integers, are there elements a, b, c, and d in S such that a + b + c + d = target? Find all unique quadruplets in the array which gives the sum of target.

Note: Elements in a quadruplet (a,b,c,d) must be in non-descending order. (ie, a ≤ b ≤ c ≤ d) The solution set must not contain duplicate quadruplets.

---

For example, given array S = {1 0 -1 0 -2 2}, and target = 0.

A solution set is:

(-1, 0, 0, 1)  
(-2, -1, 1, 2)  
(-2, 0, 0, 2)

---

### 17.1 Thoughts

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

### 17.2 Java Solution

---

```
public ArrayList<ArrayList<Integer>> fourSum(int[] num, int target) {
    Arrays.sort(num);

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

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

            while (k < l) {
                int sum = num[i] + num[j] + num[k] + num[l];

                if (sum > target) {
                    l--;
                } else if (sum < target) {
                    k++;
                } else if (sum == target) {
                    ArrayList<Integer> temp = new ArrayList<Integer>();
                    temp.add(num[i]);
                    result.add(temp);
                    hashSet.add(temp);
                }
            }
        }
    }
}
```

```
        temp.add(num[j]);
        temp.add(num[k]);
        temp.add(num[l]);

        if (!hashSet.contains(temp)) {
            hashSet.add(temp);
            result.add(temp);
        }

        k++;
        l--;
    }
}

return result;
}
```

---

Here is the hashCode method of ArrayList. It makes sure that if all elements of two lists are the same, then the hash code of the two lists will be the same. Since each element in the ArrayList is Integer, same integer has same hash code.

---

```
int hashCode = 1;
Iterator<E> i = list.iterator();
while (i.hasNext()) {
    E obj = i.next();
    hashCode = 31*hashCode + (obj==null ? 0 : obj.hashCode());
}
```

---

# 18 3Sum Closest

Given an array S of n integers, find three integers in S such that the sum is closest to a given number, target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

---

For example, given array S = {-1 2 1 -4}, and target = 1.  
The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

---

## 18.1 Analysis

This problem is similar to [2 Sum](#). This kind of problem can be solved by using a similar approach, i.e., two pointers from both left and right.

## 18.2 Java Solution

---

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

    Arrays.sort(nums);

    for (int i = 0; i < nums.length; i++) {
        int j = i + 1;
        int k = nums.length - 1;
        while (j < k) {
            int sum = nums[i] + nums[j] + nums[k];
            int diff = Math.abs(sum - target);

            if (diff == 0) return sum;

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

---

## 18 3Sum Closest

---

```
    }  
  
    return result;  
}
```

---

Time Complexity is  $O(n^2)$ .

# 19 String to Integer (atoi)

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.

## 19.1 Analysis

The following cases should be considered for this problem:

---

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

## 19.2 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;  
  
    // calculate value  
    while (str.length() > i && str.charAt(i) >= '0' && str.charAt(i) <= '9') {  
        result = result * 10 + (str.charAt(i) - '0');  
    }  
    return (int) result;  
}
```

## 19 String to Integer (atoi)

---

```
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;
}
```

---

# 20 Merge Sorted Array

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

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

## 20.1 Analysis

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

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

## 20.2 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--;
        }
    }
}
```

---

## 20.3 Java Solution 2

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

## 20 Merge Sorted Array

---

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

---

# 21 Valid Parentheses

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.

## 21.1 Analysis

A typical problem which can be solved by using a stack data structure.

## 21.2 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();
}
```

---



## 22 Longest Valid Parentheses

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2. Another example is ")()()", where the longest valid parentheses substring is "()()", which has length = 4.

### 22.1 Analysis

This problem is similar with [Valid Parentheses](#), which can be solved by using a stack.

### 22.2 Java Solution

---

```
public static int longestValidParentheses(String s) {
    Stack<int[]> stack = new Stack<int[]>();
    int result = 0;

    for(int i=0; i<=s.length()-1; i++){
        char c = s.charAt(i);
        if(c=='('){
            int[] a = {i,0};
            stack.push(a);
        }else{
            if(stack.empty()||stack.peek()[1]==1){
                int[] a = {i,1};
                stack.push(a);
            }else{
                stack.pop();
                int currentLen=0;
                if(stack.empty()){
                    currentLen = i+1;
                }else{
                    currentLen = i-stack.peek()[0];
                }
                result = Math.max(result, currentLen);
            }
        }
    }

    return result;
}
```

## 22 Longest Valid Parentheses

---

}

---

## 23 Implement strStr()

Problem:

*Implement strStr(). Returns the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.*

### 23.1 Java Solution 1 - Naive

---

```
public int strStr(String haystack, String needle) {
    if(haystack==null || needle==null)
        return 0;

    if(needle.length() == 0)
        return 0;

    for(int i=0; i<haystack.length(); i++){
        if(i + needle.length() > haystack.length())
            return -1;

        int m = i;
        for(int j=0; j<needle.length(); j++){
            if(needle.charAt(j)==haystack.charAt(m)){
                if(j==needle.length()-1)
                    return i;
                m++;
            }else{
                break;
            }
        }
    }

    return -1;
}
```

---

### 23.2 Java Solution 2 - KMP

Check out [this article](#) to understand KMP algorithm.

---

```
public int strStr(String haystack, String needle) {
    if(haystack==null || needle==null)
```

## 23 Implement strStr()

---

```
        return 0;

int h = haystack.length();
int n = needle.length();

if (n > h)
    return -1;
if (n == 0)
    return 0;

int[] next = getNext(needle);
int i = 0;

while (i <= h - n) {
    int success = 1;
    for (int j = 0; j < n; j++) {
        if (needle.charAt(j) != haystack.charAt(i + j)) {
            success = 0;
            i++;
            break;
        } else if (needle.charAt(j) == haystack.charAt(i + j)) {
            success = 1;
            i = i + j - next[j - 1];
            break;
        }
    }
    if (success == 1)
        return i;
}

return -1;
}

//calculate KMP array
public int[] getNext(String needle) {
    int[] next = new int[needle.length()];
    next[0] = 0;

    for (int i = 1; i < needle.length(); i++) {
        int index = next[i - 1];
        while (index > 0 && needle.charAt(index) != needle.charAt(i)) {
            index = next[index - 1];
        }

        if (needle.charAt(index) == needle.charAt(i)) {
            next[i] = next[i - 1] + 1;
        } else {
            next[i] = 0;
        }
    }
}
```

```
    return next;  
}
```

---



# 24 Minimum Size Subarray Sum

Given an array of  $n$  positive integers and a positive integer  $s$ , find the minimal length of a subarray of which the sum  $\geq s$ . If there isn't one, return 0 instead.

For example, given the array [2,3,1,2,4,3] and  $s = 7$ , the subarray [4,3] has the minimal length of 2 under the problem constraint.

## 24.1 Analysis

We can use 2 points to mark the left and right boundaries of the sliding window. When the sum is greater than the target, shift the left pointer; when the sum is less than the target, shift the right pointer.

## 24.2 Java Solution 1

A simple sliding window solution.

---

```
public int minSubArrayLen(int s, int[] nums) {
    if(nums==null || nums.length==1)
        return 0;

    int result = nums.length;
    int start=0;
    int sum=0;
    int i=0;
    boolean exists = false;

    while(i<=nums.length){
        if(sum>=s){
            exists=true; //mark if there exists such a subarray
            if(start==i-1){
                return 1;
            }

            result = Math.min(result, i-start);
            sum=sum-nums[start];
            start++;
        }

        }else{
            if(i==nums.length)
                break;
        }
    }
}
```

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

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

---

### 24.3 Deprecated Java Solution

This solution works but it is less readable.

```
public int minSubArrayLen(int s, int[] nums) {
    if(nums == null || nums.length == 0){
        return 0;
    }
    if(nums.length == 1 && nums[0] < s){
        return 0;
    }

    // initialize min length to be the input array length
    int result = nums.length;

    int i = 0;
    int sum = nums[0];

    for(int j=0; j<nums.length; ){
        if(i==j){
            if(nums[i]>=s){
                return 1; //if single elem is large enough
            }else{
                j++;
            }
        }else{
            if(j<nums.length){
                sum = sum + nums[j];
            }else{
                return result;
            }
        }
    }else{
        //if sum is large enough, move left cursor
        if(sum >= s){
            result = Math.min(j-i+1, result);
            sum = sum - nums[i];
            i++;
        }
    }
}
```

```
//if sum is not large enough, move right cursor
}else{
    j++;

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

return result;
}
```

---



## 25 Search Insert Position

Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order. You may assume no duplicates in the array.

Here are few examples.

---

```
[1,3,5,6], 5 -> 2
[1,3,5,6], 2 -> 1
[1,3,5,6], 7 -> 4
[1,3,5,6], 0 -> 0
```

---

### 25.1 Solution 1

Naively, we can just iterate the array and compare target with ith and (i+1)th element. Time complexity is O(n)

---

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

        if(A==null) return 0;

        if(target <= A[0]) return 0;

        for(int i=0; i<A.length-1; i++){
            if(target > A[i] && target <= A[i+1]){
                return i+1;
            }
        }

        return A.length;
    }
}
```

---

### 25.2 Solution 2

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

---

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

## 25 Search Insert Position

---

```
if(A==null||A.length==0)
    return 0;

return searchInsert(A,target,0,A.length-1);
}

public int searchInsert(int[] A, int target, int start, int end){
    int mid=(start+end)/2;

    if(target==A[mid])
        return mid;
    else if(target<A[mid])
        return start<mid?searchInsert(A,target,start,mid-1):start;
    else
        return end>mid?searchInsert(A,target,mid+1,end):(end+1);
}
}
```

---

# 26 Longest Consecutive Sequence

Given an unsorted array of integers, find the length of the longest consecutive elements sequence.

For example, given [100, 4, 200, 1, 3, 2], the longest consecutive elements sequence should be [1, 2, 3, 4]. Its length is 4.

Your algorithm should run in  $O(n)$  complexity.

## 26.1 Analysis

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

## 26.2 Java Solution

We can use a HashSet to add and remove elements. HashSet is implemented by using a hash table. Elements are not ordered. The add, remove and contains methods have constant time complexity  $O(1)$ .

```
public static int longestConsecutive(int[] num) {
    // if array is empty, return 0
    if (num.length == 0) {
        return 0;
    }

    Set<Integer> set = new HashSet<Integer>();
    int max = 1;

    for (int e : num)
        set.add(e);

    for (int e : num) {
        int left = e - 1;
        int right = e + 1;
        int count = 1;

        while (set.contains(left)) {
            count++;
            set.remove(left);
            left--;
        }
    }
}
```

```
        while (set.contains(right)) {
            count++;
            set.remove(right);
            right++;
        }

        max = Math.max(count, max);
    }

    return max;
}
```

---

After an element is checked, it should be removed from the set. Otherwise, time complexity would be  $O(mn)$  in which  $m$  is the average length of all consecutive sequences.

To clearly see the time complexity, I suggest you use some simple examples and manually execute the program. For example, given an array  $1,2,4,5,3$ , the program time is  $m$ .  $m$  is the length of longest consecutive sequence.

# 27 Valid Palindrome

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

For example, "Red rum, sir, is murder" is a palindrome, while "Programcreek is awesome" is not.

Note: Have you consider that the string might be empty? This is a good question to ask during an interview.

For the purpose of this problem, we define empty string as valid palindrome.

## 27.1 Thoughts

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

## 27.2 Java Solution 1 - Naive

---

```
public class Solution {

    public boolean isPalindrome(String s) {

        if(s == null) return false;
        if(s.length() < 2) return true;

        char[] charArray = s.toCharArray();
        int len = s.length();

        int i=0;
        int j=len-1;

        while(i<j){
            char left, right;

            while(i<len-1 && !isAlpha(left) && !isNum(left)){
                i++;
                left = charArray[i];
            }

            while(j>0 && !isAlpha(right) && !isNum(right)){
                j--;
                right = charArray[j];
            }

            if(left != right)
                return false;
        }
    }
}
```

```
        right = charArray[j];
    }

    if(i >= j)
        break;

    left = charArray[i];
    right = charArray[j];

    if(!isSame(left, right)){
        return false;
    }

    i++;
    j--;
}
return true;
}

public boolean isAlpha(char a){
    if((a >= 'a' && a <= 'z') || (a >= 'A' && a <= 'Z')){
        return true;
    }else{
        return false;
    }
}

public boolean isNum(char a){
    if(a >= '0' && a <= '9'){
        return true;
    }else{
        return false;
    }
}

public boolean isSame(char a, char b){
    if(isNum(a) && isNum(b)){
        return a == b;
    }else if(Character.toLowerCase(a) == Character.toLowerCase(b)){
        return true;
    }else{
        return false;
    }
}
```

---

### 27.3 Java Solution 2 - Using Stack

This solution removes the special characters first. (Thanks to Tia)

---

```
public boolean isPalindrome(String s) {
    s = s.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();

    int len = s.length();
    if (len < 2)
        return true;

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

    int index = 0;
    while (index < len / 2) {
        stack.push(s.charAt(index));
        index++;
    }

    if (len % 2 == 1)
        index++;

    while (index < len) {
        if (stack.empty())
            return false;

        char temp = stack.pop();
        if (s.charAt(index) != temp)
            return false;
        else
            index++;
    }

    return true;
}
```

---

### 27.4 Java Solution 3 - Using Two Pointers

In the discussion below, April and Frank use two pointers to solve this problem. This solution looks really simple.

---

```
public class ValidPalindrome {
    public static boolean isValidPalindrome(String s){
        if(s==null||s.length()==0) return false;

        s = s.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();
        System.out.println(s);
```

## 27 Valid Palindrome

---

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

return true;
}

public static void main(String[] args) {
    String str = "A man, a plan, a canal: Panama";
    System.out.println(isValidPalindrome(str));
}
}
```

---

## 28 ZigZag Conversion

The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

---

```
P A H N
A P L S I I G
Y I R
```

---

And then read line by line: "PAHNAPLSIIGYIR" Write the a method convert("PAYPALISHIRING", 3) which returns "PAHNAPLSIIGYIR".

### 28.1 Java Solution

---

```
public String convert(String s, int numRows) {
    if (numRows == 1)
        return s;

    StringBuilder sb = new StringBuilder();
    // step
    int step = 2 * numRows - 2;

    for (int i = 0; i < numRows; i++) {
        //first & last rows
        if (i == 0 || i == numRows - 1) {
            for (int j = i; j < s.length(); j = j + step) {
                sb.append(s.charAt(j));
            }
        }
        //middle rows
        } else {
            int j = i;
            boolean flag = true;
            int step1 = 2 * (numRows - 1 - i);
            int step2 = step - step1;

            while (j < s.length()) {
                sb.append(s.charAt(j));
                if (flag)
                    j = j + step1;
                else
                    j = j + step2;
                flag = !flag;
            }
        }
    }

    return sb.toString();
}
```

```
        }
    }

    return sb.toString();
}
```

---

## 29 Add Binary

Given two binary strings, return their sum (also a binary string).

For example, a = "11", b = "1", the return is "100".

### 29.1 Java Solution

Very simple, nothing special. Note how to convert a character to an int.

---

```
public String addBinary(String a, String b) {
    if(a==null || a.length()==0)
        return b;
    if(b==null || b.length()==0)
        return a;

    int pa = a.length()-1;
    int pb = b.length()-1;

    int flag = 0;
    StringBuilder sb = new StringBuilder();
    while(pa >= 0 || pb >=0){
        int va = 0;
        int vb = 0;

        if(pa >= 0){
            va = a.charAt(pa)=='0'? 0 : 1;
            pa--;
        }
        if(pb >= 0){
            vb = b.charAt(pb)=='0'? 0: 1;
            pb--;
        }

        int sum = va + vb + flag;
        if(sum >= 2){
            sb.append(String.valueOf(sum-2));
            flag = 1;
        }else{
            flag = 0;
            sb.append(String.valueOf(sum));
        }
    }

    if(flag == 1){

```

## 29 Add Binary

---

```
    sb.append("1");
}

String reversed = sb.reverse().toString();
return reversed;
}
```

---

# 30 Length of Last Word

Given a string s consists of upper/lower-case alphabets and empty space characters ' ', return the length of last word in the string. If the last word does not exist, return 0.

## 30.1 Java Solution

Very simple question. We just need a flag to mark the start of letters from the end. If a letter starts and the next character is not a letter, return the length.

---

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

    int result = 0;
    int len = s.length();

    boolean flag = false;
    for(int i=len-1; i>=0; i--){
        char c = s.charAt(i);
        if((c>='a' && c<='z') || (c>='A' && c<='Z')){
            flag = true;
            result++;
        }else{
            if(flag)
                return result;
        }
    }

    return result;
}
```

---



# 31 Triangle

Given a triangle, find the minimum path sum from top to bottom. Each step you may move to adjacent numbers on the row below.

For example, given the following triangle

---

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

---

The minimum path sum from top to bottom is 11 (i.e.,  $2 + 3 + 5 + 1 = 11$ ).

Note: Bonus point if you are able to do this using only  $O(n)$  extra space, where  $n$  is the total number of rows in the triangle.

## 31.1 Top-Down Approach (Wrong Answer!)

This solution gets wrong answer! I will try to make it work later.

---

```
public class Solution {  
    public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {  
  
        int[] temp = new int[triangle.size()];  
        int minTotal = Integer.MAX_VALUE;  
  
        for(int i=0; i < temp.length; i++){  
            temp[i] = Integer.MAX_VALUE;  
        }  
  
        if (triangle.size() == 1) {  
            return Math.min(minTotal, triangle.get(0).get(0));  
        }  
  
        int first = triangle.get(0).get(0);  
  
        for (int i = 0; i < triangle.size() - 1; i++) {  
            for (int j = 0; j <= i; j++) {  
  
                int a, b;  
  
                if(i==0 && j==0){
```

```
a = first + triangle.get(i + 1).get(j);
b = first + triangle.get(i + 1).get(j + 1);

}else{
    a = temp[j] + triangle.get(i + 1).get(j);
    b = temp[j] + triangle.get(i + 1).get(j + 1);

}

temp[j] = Math.min(a, temp[j]);
temp[j + 1] = Math.min(b, temp[j + 1]);
}
}

for (int e : temp) {
    if (e < minTotal)
        minTotal = e;
}

return minTotal;
}
}
```

---

## 31.2 Bottom-Up (Good Solution)

We can actually start from the bottom of the triangle.

---

```
public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {
    int[] total = new int[triangle.size()];
    int l = triangle.size() - 1;

    for (int i = 0; i < triangle.get(l).size(); i++) {
        total[i] = triangle.get(l).get(i);
    }

    // iterate from last second row
    for (int i = triangle.size() - 2; i >= 0; i--) {
        for (int j = 0; j < triangle.get(i + 1).size() - 1; j++) {
            total[j] = triangle.get(i).get(j) + Math.min(total[j], total[j + 1]);
        }
    }

    return total[0];
}
```

---

## 32 Contains Duplicate

Given an array of integers, find if the array contains any duplicates. Your function should return true if any value appears at least twice in the array, and it should return false if every element is distinct.

### 32.1 Java Solution

---

```
public boolean containsDuplicate(int[] nums) {  
    if(nums==null || nums.length==0)  
        return false;  
  
    HashSet<Integer> set = new HashSet<Integer>();  
    for(int i: nums){  
        if(!set.add(i)){  
            return true;  
        }  
    }  
  
    return false;  
}
```

---



## 33 Contains Duplicate II

Given an array of integers and an integer  $k$ , return true if and only if there are two distinct indices  $i$  and  $j$  in the array such that  $\text{nums}[i] = \text{nums}[j]$  and the difference between  $i$  and  $j$  is at most  $k$ .

### 33.1 Java Solution 1

---

```
public boolean containsNearbyDuplicate(int[] nums, int k) {
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    int min = Integer.MAX_VALUE;

    for(int i=0; i<nums.length; i++){
        if(map.containsKey(nums[i])){
            int preIndex = map.get(nums[i]);
            int gap = i-preIndex;
            min = Math.min(min, gap);
        }
        map.put(nums[i], i);
    }

    if(min <= k){
        return true;
    }else{
        return false;
    }
}
```

---

### 33.2 Java Solution 2 - Simplified

---

```
public boolean containsNearbyDuplicate(int[] nums, int k) {
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

    for(int i=0; i<nums.length; i++){
        if(map.containsKey(nums[i])){
            int pre = map.get(nums[i]);
            if(i-pre<=k)
                return true;
        }
    }
}
```

---

### 33 Contains Duplicate II

---

```
    map.put(nums[i], i);
}

return false;
}
```

---

## 34 Contains Duplicate III

Given an array of integers, find out whether there are two distinct indices  $i$  and  $j$  in the array such that the difference between  $\text{nums}[i]$  and  $\text{nums}[j]$  is at most  $t$  and the difference between  $i$  and  $j$  is at most  $k$ .

### 34.1 Java Solution 1 - Simple

UPDATED on 5/8/2016.

This solution simple. Its time complexity is  $O(n \log(k))$ .

---

```
public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
    if(nums==null||nums.length<2||k<0||t<0)
        return false;

    TreeSet<Long> set = new TreeSet<Long>();
    for(int i=0; i<nums.length; i++){
        long curr = (long) nums[i];

        long leftBoundary = (long) curr-t;
        long rightBoundary = (long) curr+t+1; //right boundary is exclusive, so
        +1
        SortedSet<Long> sub = set.subSet(leftBoundary, rightBoundary);
        if(sub.size()>0)
            return true;

        set.add(curr);

        if(i>=k){
            set.remove((long)nums[i-k]);
        }
    }

    return false;
}
```

---

## 34.2 Java Solution 2 - Deprecated

1 2 4 5

$\text{floor}(3) = 2$

$\text{ceiling}(3) = 4$

---

```
public boolean containsNearbyAlmostDuplicate(int[] nums, int k, int t) {
    if (k < 1 || t < 0)
        return false;

    TreeSet<Integer> set = new TreeSet<Integer>();

    for (int i = 0; i < nums.length; i++) {
        int c = nums[i];
        if ((set.floor(c) != null && c <= set.floor(c) + t)
            || (set.ceiling(c) != null && c >= set.ceiling(c) - t))
            return true;

        set.add(c);

        if (i >= k)
            set.remove(nums[i - k]);
    }

    return false;
}
```

---

## 35 Remove Duplicates from Sorted Array

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

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

### 35.1 Thoughts

The problem is pretty straightforward. It returns the length of array with unique elements, but the original array need to be changed also. This problem should be reviewed with [Remove Duplicates from Sorted Array II](#).

### 35.2 Solution 1

---

```
// Manipulate original array
public static int removeDuplicatesNaive(int[] A) {
    if (A.length < 2)
        return A.length;

    int j = 0;
    int i = 1;

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

    return j + 1;
}
```

---

This method returns the number of unique elements, but does not change the original array correctly. For example, if the input array is 1, 2, 2, 3, 3, the array will be changed to 1, 2, 3, 3, 3. The correct result should be 1, 2, 3. Because array's size can

## 35 Remove Duplicates from Sorted Array

---

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

### 35.3 Solution 2

---

```
// Create an array with all unique elements
public static int[] removeDuplicates(int[] A) {
    if (A.length < 2)
        return A;

    int j = 0;
    int i = 1;

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

    int[] B = Arrays.copyOf(A, j + 1);

    return B;
}

public static void main(String[] args) {
    int[] arr = { 1, 2, 2, 3, 3 };
    arr = removeDuplicates(arr);
    System.out.println(arr.length);
}
```

---

In this method, a new array is created and returned.

### 35.4 Solution 3

If we only want to count the number of unique elements, the following method is good enough.

---

```
// Count the number of unique elements
public static int countUnique(int[] A) {
    int count = 0;
    for (int i = 0; i < A.length - 1; i++) {
        if (A[i] == A[i + 1])
            count++;
    }
}
```

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

public static void main(String[] args) {
    int[] arr = { 1, 2, 2, 3, 3 };
    int size = countUnique(arr);
    System.out.println(size);
}
```

---



## 36 Remove Duplicates from Sorted Array II

Follow up for "Remove Duplicates": What if duplicates are allowed at most twice?

For example, given sorted array A = [1,1,1,2,2,3], your function should return length = 5, and A is now [1,1,2,2,3].

So this problem also requires in-place array manipulation.

### 36.1 Java Solution 1

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

---

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

        int pre = A[0];
        boolean flag = false;
        int count = 0;

        // index for updating
        int o = 1;

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

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

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

```
    }

    return A.length - count;
}

}
```

---

## 36.2 Java Solution 2

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

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

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

        return prev + 1;
    }
}
```

---

## 37 Remove Element

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

### 37.1 Java Solution

This problem can be solve by using two indices.

---

```
public int removeElement(int[] A, int elem) {  
    int i=0;  
    int j=0;  
  
    while(j < A.length){  
        if(A[j] != elem){  
            A[i] = A[j];  
            i++;  
        }  
  
        j++;  
    }  
  
    return i;  
}
```

---



## 38 Move Zeroes

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

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

### 38.1 Java Solution 1

---

```
public void moveZeroes(int[] nums) {
    int m=-1;

    for(int i=0; i<nums.length; i++){
        if(nums[i]==0){
            if(m==-1 || nums[m]!=0){
                m=i;
            }
        }else{
            if(m!=-1){
                int temp = nums[i];
                nums[i]=nums[m];
                nums[m]=temp;
                m++;
            }
        }
    }
}
```

---

### 38.2 Java Solution 2 - Simple

Actually, we can use the similar code that is used to solve [Remove Duplicates from Sorted Array I, II](#), [Remove Element](#). We can use almost identical code to solve those problems!

---

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

    while(j<nums.length){
        if(nums[j]==0){
```

### 38 Move Zeroes

---

```
        j++;
    }else{
        nums[i]=nums[j];
        i++;
        j++;
    }
}

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


---


```

# 39 Longest Substring Without Repeating Characters

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

## 39.1 Java Solution 1

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

---

```
public int lengthOfLongestSubstring(String s) {
    if(s==null)
        return 0;
    boolean[] flag = new boolean[256];

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

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

## 39 Longest Substring Without Repeating Characters

---

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

    return result;
}
```

---

### 39.2 Java Solution 2

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

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

```
public static int lengthOfLongestSubstring(String s) {
    if(s==null)
        return 0;
    char[] arr = s.toCharArray();
    int pre = 0;

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

    for (int i = 0; i < arr.length; i++) {
        if (!map.containsKey(arr[i])) {
            map.put(arr[i], i);
        } else {
            pre = Math.max(pre, map.size());
            i = map.get(arr[i]);
            map.clear();
        }
    }

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

---

### 39.3 Java Solution 3

```
public int lengthOfLongestSubstring(String s) {
    if(s==null){
        return 0;
    }

    int max = 0;

    HashMap<Character, Integer> map = new HashMap<Character, Integer>();
    int start=0;
```

```
for(int i=0; i<s.length(); i++){
    char c = s.charAt(i);
    if(map.containsKey(c)){
        max = Math.max(max, map.size());
        while(map.containsKey(c)){
            map.remove(s.charAt(start));
            start++;
        }
        map.put(c, i);
    }else{
        map.put(c, i);
    }
}
max = Math.max(max, map.size());
return max;
}
```

---



# 40 Longest Substring Which Contains 2 Unique Characters

This is a problem asked by Google.

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

## 40.1 Longest Substring Which Contains 2 Unique Characters

In this solution, a hashmap is used to track the unique elements in the map. When a third character is added to the map, the left pointer needs to move right.

You can use "abac" to walk through this solution.

---

```
public int lengthOfLongestSubstringTwoDistinct(String s) {
    int max=0;
    HashMap<Character, Integer> map = new HashMap<Character, Integer>();
    int start=0;

    for(int i=0; i<s.length(); i++){
        char c = s.charAt(i);
        if(map.containsKey(c)){
            map.put(c, map.get(c)+1);
        }else{
            map.put(c, 1);
        }

        if(map.size()>2){
            max = Math.max(max, i-start);

            while(map.size()>2){
                char t = s.charAt(start);
                int count = map.get(t);
                if(count>1){
                    map.put(t, count-1);
                }else{
                    map.remove(t);
                }
                start++;
            }
        }
    }
}
```

```
        }
    }

    max = Math.max(max, s.length()-start);

    return max;
}
```

---

Now if this question is extended to be "the longest substring that contains k unique characters", what should we do?

## 40.2 Solution for K Unique Characters

The following solution is corrected. Given "abcadacacacaca" and 3, it returns "cadcacacaca".

```
public int lengthOfLongestSubstringKDistinct(String s, int k) {
    int max=0;
    HashMap<Character, Integer> map = new HashMap<Character, Integer>();
    int start=0;

    for(int i=0; i<s.length(); i++){
        char c = s.charAt(i);
        if(map.containsKey(c)){
            map.put(c, map.get(c)+1);
        }else{
            map.put(c,1);
        }

        if(map.size()>k){
            max = Math.max(max, i-start);

            while(map.size()>k){
                char t = s.charAt(start);
                int count = map.get(t);
                if(count>1){
                    map.put(t, count-1);
                }else{
                    map.remove(t);
                }
                start++;
            }
        }
    }

    max = Math.max(max, s.length()-start);

    return max;
}
```

---

Time is  $O(n)$ .



# 41 Substring with Concatenation of All Words

You are given a string,  $s$ , and a list of words,  $\text{words}$ , that are all of the same length. Find all starting indices of substring(s) in  $s$  that is a concatenation of each word in  $\text{words}$  exactly once and without any intervening characters.

For example, given:  $s=\text{"barfoothefoobarman"}$  &  $\text{words}=[\text{"foo", "bar"}]$ , return [0,9].

## 41.1 Analysis

This problem is similar (almost the same) to [Longest Substring Which Contains 2 Unique Characters](#).

Since each word in the dictionary has the same length, each of them can be treated as a single character.

## 41.2 Java Solution

---

```
public List<Integer> findSubstring(String s, String[] words) {
    ArrayList<Integer> result = new ArrayList<Integer>();
    if(s==null||s.length()==0||words==null||words.length==0){
        return result;
    }

    //frequency of words
    HashMap<String, Integer> map = new HashMap<String, Integer>();
    for(String w: words){
        if(map.containsKey(w)){
            map.put(w, map.get(w)+1);
        }else{
            map.put(w, 1);
        }
    }

    int len = words[0].length();

    for(int j=0; j<len; j++){
        HashMap<String, Integer> currentMap = new HashMap<String, Integer>();
        int start = j;//start index of start
        int count = 0;//count total qualified words so far
```

## 41 Substring with Concatenation of All Words

---

```
for(int i=j; i<=s.length()-len; i=i+len){
    String sub = s.substring(i, i+len);
    if(map.containsKey(sub)){
        //set frequency in current map
        if(currentMap.containsKey(sub)){
            currentMap.put(sub, currentMap.get(sub)+1);
        }else{
            currentMap.put(sub, 1);
        }

        count++;

        while(currentMap.get(sub)>map.get(sub)){
            String left = s.substring(start, start+len);
            currentMap.put(left, currentMap.get(left)-1);

            count--;
            start = start + len;
        }

        if(count==words.length){
            result.add(start); //add to result

            //shift right and reset currentMap, count & start point
            String left = s.substring(start, start+len);
            currentMap.put(left, currentMap.get(left)-1);
            count--;
            start = start + len;
        }
    }else{
        currentMap.clear();
        start = i+len;
        count = 0;
    }
}

return result;
}
```

---

## 42 Minimum Window Substring

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

For example, S = "ADOBECODEBANC", T = "ABC", Minimum window is "BANC".

### 42.1 Java Solution

---

```
public String minWindow(String s, String t) {
    if(t.length()>s.length())
        return "";
    String result = "";

    //character counter for t
    HashMap<Character, Integer> target = new HashMap<Character, Integer>();
    for(int i=0; i<t.length(); i++){
        char c = t.charAt(i);
        if(target.containsKey(c)){
            target.put(c,target.get(c)+1);
        }else{
            target.put(c,1);
        }
    }

    // character counter for s
    HashMap<Character, Integer> map = new HashMap<Character, Integer>();
    int left = 0;
    int minLen = s.length()+1;

    int count = 0; // the total of mapped characters

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

        if(target.containsKey(c)){
            if(map.containsKey(c)){
                if(map.get(c)<target.get(c)){
                    count++;
                }
                map.put(c,map.get(c)+1);
            }else{
                map.put(c,1);
                count++;
            }
        }
    }
}
```

```
        }
    }

    if(count == t.length()){
        char sc = s.charAt(left);
        while (!map.containsKey(sc) || map.get(sc) > target.get(sc)) {
            if (map.containsKey(sc) && map.get(sc) > target.get(sc))
                map.put(sc, map.get(sc) - 1);
            left++;
            sc = s.charAt(left);
        }

        if (i - left + 1 < minLen) {
            result = s.substring(left, i + 1);
            minLen = i - left + 1;
        }
    }
}

return result;
}
```

---

## 43 Reverse Words in a String

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

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

### 43.1 Java Solution

This problem is pretty straightforward. We first split the string to words array, and then iterate through the array and add each element to a new string. Note: StringBuilder should be used to avoid creating too many Strings. If the string is very long, using String is not scalable since `String` is immutable and too many objects will be created and garbage collected.

---

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

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

---



# 44 Find Minimum in Rotated Sorted Array

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

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

## 44.1 Analysis

This problem is a binary search and the key is breaking the array to two parts, so that we only need to work on half of the array each time.

If we pick the middle element, we can compare the middle element with the leftmost (or rightmost) element. If the middle element is less than leftmost, the left half should be selected; if the middle element is greater than the leftmost (or rightmost), the right half should be selected. Using recursion or iteration, this problem can be solved in time  $\log(n)$ .

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

## 44.2 Java Solution 1 - Recursion

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

---

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

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

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

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

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

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

---

### 44.3 Java Solution 2 - Iteration

```
public int findMin(int[] nums) {
    if(nums.length==1)
        return nums[0];

    int left=0;
    int right=nums.length-1;

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

    while(left <= right){
        if(right-left==1){
            return nums[right];
        }

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

        if(nums[m] > nums[right])
            left = m;
        else
            right = m;
    }

    return nums[left];
}
```

---

# 45 Find Minimum in Rotated Sorted Array II

## 45.1 Problem

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

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

## 45.2 Java Solution

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

---

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

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

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

## 45 Find Minimum in Rotated Sorted Array II

---

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

---

## 46 Search in Rotated Sorted Array

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

You are given a target value to search. If found in the array return its index, otherwise return -1. You may assume no duplicate exists in the array.

### 46.1 Java Solution 1- Recursive

---

```
public int search(int[] nums, int target) {
    return binarySearch(nums, 0, nums.length-1, target);
}

public int binarySearch(int[] nums, int left, int right, int target){
    if(left>right)
        return -1;

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

    if(target == nums[mid])
        return mid;

    if(nums[left] <= nums[mid]){
        if(nums[left]<=target && target<nums[mid]){
            return binarySearch(nums, left, mid-1, target);
        }else{
            return binarySearch(nums, mid+1, right, target);
        }
    }else {
        if(nums[mid]<target&& target<=nums[right]){
            return binarySearch(nums, mid+1, right, target);
        }else{
            return binarySearch(nums, left, mid-1, target);
        }
    }
}
```

---

### 46.2 Java Solution 2 - Iterative

---

```
public int search(int[] nums, int target) {
```

## 46 Search in Rotated Sorted Array

---

```
int left = 0;
int right= nums.length-1;

while(left<=right){
    int mid = left + (right-left)/2;
    if(target==nums[mid])
        return mid;

    if(nums[left]<=nums[mid]){
        if(nums[left]<=target&& target<nums[mid]){
            right=mid-1;
        }else{
            left=mid+1;
        }
    }else{
        if(nums[mid]<target&& target<=nums[right]){
            left=mid+1;
        }else{
            right=mid-1;
        }
    }
}

return -1;
}
```

---

## 47 Search in Rotated Sorted Array II

Follow up for "Search in Rotated Sorted Array": what if duplicates are allowed? Write a function to determine if a given target is in the array.

### 47.1 Java Solution

---

```
public boolean search(int[] nums, int target) {
    int left=0;
    int right=nums.length-1;

    while(left<=right){
        int mid = (left+right)/2;
        if(nums[mid]==target)
            return true;

        if(nums[left]<nums[mid]){
            if(nums[left]<=target&& target<nums[mid]){
                right=mid-1;
            }else{
                left=mid+1;
            }
        }else if(nums[left]>nums[mid]){
            if(nums[mid]<target&&target<=nums[right]){
                left=mid+1;
            }else{
                right=mid-1;
            }
        }else{
            left++;
        }
    }

    return false;
}
```

---



# 48 Min Stack

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

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

## 48.1 Analysis

UPDATED ON 6/17/2015

To make constant time of getMin(), we need to keep track of the minimum element for each element in the stack.

## 48.2 Java Solution

Define a node class that holds element value, min value, and pointer to elements below it.

---

```
class Node {  
    int value;  
    int min;  
    Node next;  
  
    Node(int x) {  
        value = x;  
        next = null;  
        min = x;  
    }  
}
```

---

```
class MinStack {  
    Node head;  
  
    public void push(int x) {  
        if (head == null) {  
            head = new Node(x);  
        } else {  
            Node temp = new Node(x);  
            temp.min = Math.min(head.min, x);  
            temp.next = head;  
        }  
    }  
}
```

```
        head = temp;
    }
}

public void pop() {
    if (head == null)
        return;
    head = head.next;
}

public int top() {
    if (head == null)
        return Integer.MAX_VALUE;

    return head.value;
}

public int getMin() {
    if (head == null)
        return Integer.MAX_VALUE;

    return head.min;
}
}
```

---

# 49 Majority Element

Given an array of size  $n$ , find the majority element. The majority element is the element that appears more than  $\lfloor n/2 \rfloor$  times. (assume that the array is non-empty and the majority element always exist in the array.)

## 49.1 Java Solution 1 - Naive

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

---

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

        Arrays.sort(num);

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

        return 0;
    }
}
```

---

## 49.2 Java Solution 2 - Much Simpler

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

## 49 Majority Element

---

```
public int majorityElement(int[] num) {  
    if (num.length == 1) {  
        return num[0];  
    }  
  
    Arrays.sort(num);  
    return num[num.length / 2];  
}
```

---

### 49.3 Java Solution 3 - Linear Time Majority Vote Algorithm

---

```
public int majorityElement(int[] nums) {  
    int result = 0, count = 0;  
  
    for(int i = 0; i<nums.length; i++ ) {  
        if(count == 0){  
            result = nums[ i ];  
            count = 1;  
        }else if(result == nums[i]){  
            count++;  
        }else{  
            count--;  
        }  
    }  
  
    return result;  
}
```

---

# 50 Majority Element II

Given an integer array of size  $n$ , find all elements that appear more than  $\lfloor n/3 \rfloor$  times.  
The algorithm should run in linear time and in  $O(1)$  space.

## 50.1 Java Solution 1 - Using a Counter

Time =  $O(n)$  and Space =  $O(n)$

---

```
public List<Integer> majorityElement(int[] nums) {
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    for(int i: nums){
        if(map.containsKey(i)){
            map.put(i, map.get(i)+1);
        }else{
            map.put(i, 1);
        }
    }

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

    for(Map.Entry<Integer, Integer> entry: map.entrySet()){
        if(entry.getValue() > nums.length/3){
            result.add(entry.getKey());
        }
    }

    return result;
}
```

---

## 50.2 Java Solution 2

Time =  $O(n)$  and Space =  $O(1)$

Check out [Majority Element I](#).

---

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

    Integer n1=null, n2=null;
    int c1=0, c2=0;

    for(int i: nums){
```

## 50 Majority Element II

---

```
if(n1!=null && i==n1.intValue()){
    c1++;
}else if(n2!=null && i==n2.intValue()){
    c2++;
}else if(c1==0){
    c1=1;
    n1=i;
}else if(c2==0){
    c2=1;
    n2=i;
}else{
    c1--;
    c2--;
}
}

c1=c2=0;

for(int i: nums){
    if(i==n1.intValue()){
        c1++;
    }else if(i==n2.intValue()){
        c2++;
    }
}

if(c1>nums.length/3)
    result.add(n1);
if(c2>nums.length/3)
    result.add(n2);

return result;
}
```

---

# 51 Bulls and Cows

You are playing the following Bulls and Cows game with your friend: You write down a number and ask your friend to guess what the number is. Each time your friend makes a guess, you provide a hint that indicates how many digits in said guess match your secret number exactly in both digit and position (called "bulls") and how many digits match the secret number but locate in the wrong position (called "cows"). Your friend will use successive guesses and hints to eventually derive the secret number.

For example: Secret number: "1807" Friend's guess: "7810"

Hint: 1 bull and 3 cows. (The bull is 8, the cows are 0, 1 and 7.) Write a function to return a hint according to the secret number and friend's guess, use A to indicate the bulls and B to indicate the cows. In the above example, your function should return "1A3B".

## 51.1 Java Solution 1 - Using HashMap

---

```
public String getHint(String secret, String guess) {
    int countBull=0;
    int countCow=0;

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

    //check bull
    for(int i=0; i<secret.length(); i++){
        char c1 = secret.charAt(i);
        char c2 = guess.charAt(i);

        if(c1==c2){
            countBull++;
        }else{
            if(map.containsKey(c1)){
                int freq = map.get(c1);
                freq++;
                map.put(c1, freq);
            }else{
                map.put(c1, 1);
            }
        }
    }

    //check cow
    for(int i=0; i<secret.length(); i++){
        char c1 = secret.charAt(i);
        char c2 = guess.charAt(i);

        if(c1!=c2){
            if(map.containsKey(c1)){
                int freq = map.get(c1);
                freq--;
                map.put(c1, freq);
            }else{
                map.put(c1, -1);
            }

            if(map.containsKey(c2)){
                int freq = map.get(c2);
                freq++;
                map.put(c2, freq);
            }else{
                map.put(c2, 1);
            }
        }
    }
}
```

## 51 Bulls and Cows

---

```
for(int i=0; i<secret.length(); i++){
    char c1 = secret.charAt(i);
    char c2 = guess.charAt(i);

    if(c1!=c2){
        if(map.containsKey(c2)){
            countCow++;
            if(map.get(c2)==1){
                map.remove(c2);
            }else{
                int freq = map.get(c2);
                freq--;
                map.put(c2, freq);
            }
        }
    }
}

return countBull+"A"+countCow+"B";
}
```

---

## 51.2 Java Solution 2 - Using an Array

Since the secret and guess only contain numbers and there are at most 10 possible digits, we can use two arrays to track the frequency of each digits in secret and guess.

---

```
public String getHint(String secret, String guess) {
    int countBull=0;
    int countCow =0;
    int[] arr1 = new int[10];
    int[] arr2 = new int[10];

    for(int i=0; i<secret.length(); i++){
        char c1 = secret.charAt(i);
        char c2 = guess.charAt(i);

        if(c1==c2)
            countBull++;
        else{
            arr1[c1-'0']++;
            arr2[c2-'0']++;
        }
    }

    for(int i=0; i<10; i++){
        countCow += Math.min(arr1[i], arr2[i]);
    }
}
```

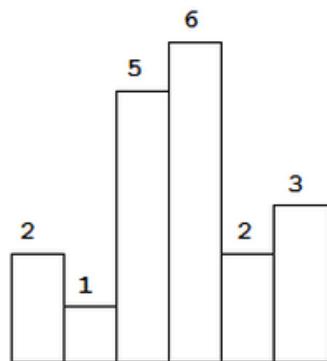
```
    return countBull+"A"+countCow+"B";
}
```

---

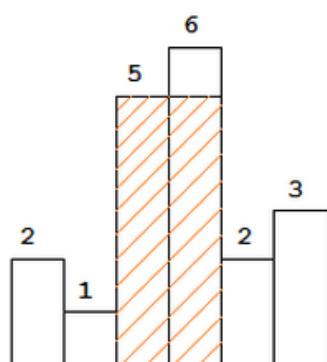


## 52 Largest Rectangle in Histogram

Given  $n$  non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.



Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].



For example, given height = [2,1,5,6,2,3], return 10.

### 52.1 Analysis

The key to solve this problem is to maintain a stack to store bars' indexes. The stack only keeps the increasing bars.

## 52.2 Java Solution

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

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

    int max = 0;
    int i = 0;

    while (i < height.length) {
        //push index to stack when the current height is larger than the previous
        //one
        if (stack.isEmpty() || height[i] >= height[stack.peek()]) {
            stack.push(i);
            i++;
        } else {
            //calculate max value when the current height is less than the previous
            //one
            int p = stack.pop();
            int h = height[p];
            int w = stack.isEmpty() ? i : i - stack.peek() - 1;
            max = Math.max(h * w, max);
        }
    }

    while (!stack.isEmpty()) {
        int p = stack.pop();
        int h = height[p];
        int w = stack.isEmpty() ? i : i - stack.peek() - 1;
        max = Math.max(h * w, max);
    }

    return max;
}
```

---

# 53 Longest Common Prefix

## 53.1 Problem

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

## 53.2 Analysis

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

## 53.3 Java Solution

---

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

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

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

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

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

---



# 54 Largest Number

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

For example, given [3, 30, 34, 5, 9], the largest formed number is 9534330. (Note: The result may be very large, so you need to return a string instead of an integer.)

## 54.1 Analysis

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

## 54.2 Java Solution

---

```
public String largestNumber(int[] nums) {
    String[] strs = new String[nums.length];
    for(int i=0; i<nums.length; i++){
        strs[i] = String.valueOf(nums[i]);
    }

    Arrays.sort(strs, new Comparator<String>(){
        public int compare(String s1, String s2){
            String leftRight = s1+s2;
            String rightLeft = s2+s1;
            return -leftRight.compareTo(rightLeft);
        }
    });

    StringBuilder sb = new StringBuilder();
    for(String s: strs){
        sb.append(s);
    }

    while(sb.charAt(0)=='0' && sb.length()>1){
        sb.deleteCharAt(0);
    }

    return sb.toString();
}
```

---



## 55 Simplify Path

Given an absolute path for a file (Unix-style), simplify it.

For example,

---

```
path = "/home/", => "/home"
path = "/a/./b/../../c/", => "/c"
path = "/../", => "/"
path = "/home//foo/", => "/home/foo"
```

---

### 55.1 Java Solution

---

```
public String simplifyPath(String path) {
    Stack<String> stack = new Stack<String>();

    //stack.push(path.substring(0,1));

    while(path.length() > 0 && path.charAt(path.length()-1) == '/') {
        path = path.substring(0, path.length()-1);
    }

    int start = 0;
    for(int i=1; i<path.length(); i++){
        if(path.charAt(i) == '/'){
            stack.push(path.substring(start, i));
            start = i;
        }else if(i==path.length()-1){
            stack.push(path.substring(start));
        }
    }

    LinkedList<String> result = new LinkedList<String>();
    int back = 0;
    while(!stack.isEmpty()){
        String top = stack.pop();

        if(top.equals(".")) || top.equals("/")){
            //nothing
        }else if(top.equals("../")){
            back++;
        }else{
            if(back > 0){


```

## 55 Simplify Path

---

```
        back--;
    }else{
        result.push(top);
    }
}

//if empty, return "/"
if(result.isEmpty()){
    return "/";
}

StringBuilder sb = new StringBuilder();
while(!result.isEmpty()){
    String s = result.pop();
    sb.append(s);
}

return sb.toString();
}
```

---

# 56 Compare Version Numbers

## 56.1 Problem

Compare two version numbers `version1` and `version2`. If `version1 > version2` return `1`, if `version1 < version2` return `-1`, otherwise return `0`. You may assume that the version strings are non-empty and contain only digits and the `.` character. The `.` character does not represent a decimal point and is used to separate number sequences. Here is an example of version numbers ordering:

---

`0.1 < 1.1 < 1.2 < 13.37`

---

## 56.2 Java Solution

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

```
public int compareVersion(String version1, String version2) {  
    String[] arr1 = version1.split("\\\\.");  
    String[] arr2 = version2.split("\\\\.");  
  
    int i=0;  
    while(i<arr1.length || i<arr2.length){  
        if(i<arr1.length && i<arr2.length){  
            if(Integer.parseInt(arr1[i]) < Integer.parseInt(arr2[i])){  
                return -1;  
            } else if(Integer.parseInt(arr1[i]) > Integer.parseInt(arr2[i])){  
                return 1;  
            }  
        } else if(i<arr1.length){  
            if(Integer.parseInt(arr1[i]) != 0){  
                return 1;  
            }  
        } else if(i<arr2.length){  
            if(Integer.parseInt(arr2[i]) != 0){  
                return -1;  
            }  
        }  
        i++;  
    }  
  
    return 0;  
}
```

## 56 Compare Version Numbers

---

}

---

## 57 Gas Station

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

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

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

### 57.1 Analysis

To solve this problem, we need to understand and use the following 2 facts: 1) if the sum of gas  $\geq$  the sum of cost, then the circle can be completed. 2) if A can not reach C in the sequence of A->B->C, then B can not make it either.

Proof of fact 2:

---

If  $\text{gas}[A] < \text{cost}[A]$ , then A can not even reach B.

So to reach C from A,  $\text{gas}[A] \geq \text{cost}[A]$ .

Given that A can not reach C, we have  $\text{gas}[A] + \text{gas}[B] < \text{cost}[A] + \text{cost}[B]$ , and  $\text{gas}[A] \geq \text{cost}[A]$ ,

Therefore,  $\text{gas}[B] < \text{cost}[B]$ , i.e., B can not reach C.

---

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

### 57.2 Java Solution

---

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

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

## 57 Gas Station

---

```
int remaining = gas[i] - cost[i];

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

if (total >= 0){
    return start;
}else{
    return -1;
}
}
```

---

## 58 Pascal's Triangle

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

---

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

---

### 58.1 Java Solution

---

```
public ArrayList<ArrayList<Integer>> generate(int numRows) {  
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();  
    if (numRows <= 0)  
        return result;  
  
    ArrayList<Integer> pre = new ArrayList<Integer>();  
    pre.add(1);  
    result.add(pre);  
  
    for (int i = 2; i <= numRows; i++) {  
        ArrayList<Integer> cur = new ArrayList<Integer>();  
  
        cur.add(1); //first  
        for (int j = 0; j < pre.size() - 1; j++) {  
            cur.add(pre.get(j) + pre.get(j + 1)); //middle  
        }  
        cur.add(1); //last  
  
        result.add(cur);  
        pre = cur;  
    }  
  
    return result;  
}
```

---



# 59 Pascal's Triangle II

Given an index k, return the kth row of the Pascal's triangle. For example, when k = 3, the row is [1,3,3,1].

## 59.1 Analysis

This problem is related to [Pascal's Triangle](#) which gets all rows of Pascal's triangle. In this problem, only one row is required to return.

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ 1 & 4 & 6 & 4 & 1 & & \\ 1 & 5 & 10 & 10 & 5 & 1 & \end{array}$$

## 59.2 Java Solution

---

```
public List<Integer>getRow(int rowIndex) {
    ArrayList<Integer> result = new ArrayList<Integer>();

    if (rowIndex < 0)
        return result;

    result.add(1);
    for (int i = 1; i <= rowIndex; i++) {
        for (int j = result.size() - 2; j >= 0; j--) {
            result.set(j + 1, result.get(j) + result.get(j + 1));
        }
        result.add(1);
    }
    return result;
}
```

---



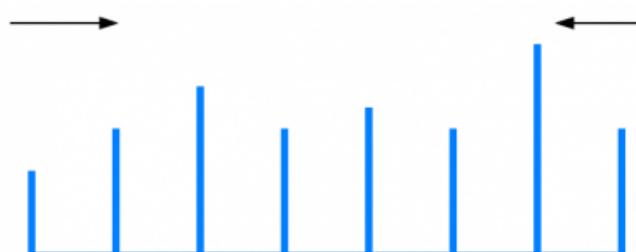
# 60 Container With Most Water

## 60.1 Problem

Given  $n$  non-negative integers  $a_1, a_2, \dots, a_n$ , where each represents a point at coordinate  $(i, a_i)$ .  $n$  vertical lines are drawn such that the two endpoints of line  $i$  is at  $(i, a_i)$  and  $(i, 0)$ . Find two lines, which together with x-axis forms a container, such that the container contains the most water.

## 60.2 Analysis

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



## 60.3 Java Solution

---

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

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

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

## 60 Container With Most Water

---

```
    if (height[left] < height[right])
        left++;
    else
        right--;
}

return max;
}
```

---

# 61 Candy

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

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

What is the minimum candies you must give?

## 61.1 Analysis

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

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

This problem is similar to Trapping Rain Water.

## 61.2 Java Solution

---

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

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

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

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

## 61 Candy

---

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

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

return result;
}
```

---

# 62 Trapping Rain Water

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

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

## 62.1 Analysis

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

## 62.2 Java Solution

---

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

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

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

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

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

## 62 Trapping Rain Water

---

```
        right[i]=height[i];
        max = height[i];
    }
}

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

return result;
}
```

---

# 63 Count and Say

## 63.1 Problem

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

---

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

---

Given an integer n, generate the nth sequence.

## 63.2 Java Solution

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

---

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

    String result = "1";
    int i = 1;

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

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

### 63 Count and Say

---

```
    return result;  
}
```

---

# 64 Search for a Range

Given a sorted array of integers, find the starting and ending position of a given target value. Your algorithm's runtime complexity must be in the order of  $O(\log n)$ . If the target is not found in the array, return [-1, -1]. For example, given [5, 7, 7, 8, 8, 10] and target value 8, return [3, 4].

## 64.1 Analysis

Based on the requirement of  $O(\log n)$ , this is a binary search problem apparently.

## 64.2 Java Solution

---

```
public int[] searchRange(int[] nums, int target) {
    if(nums == null || nums.length == 0){
        return null;
    }

    int[] arr= new int[2];
    arr[0]=-1;
    arr[1]=-1;

    binarySearch(nums, 0, nums.length-1, target, arr);

    return arr;
}

public void binarySearch(int[] nums, int left, int right, int target, int[]
arr){
    if(right<left)
        return;

    if(nums[left]==nums[right] && nums[left]==target){
        arr[0]=left;
        arr[1]=right;
        return;
    }

    int mid = left+(right-left)/2;
```

## 64 Search for a Range

---

```
if(nums[mid]<target){
    binarySearch(nums, mid+1, right, target, arr);
}else if(nums[mid]>target){
    binarySearch(nums, left, mid-1, target, arr);
}else{
    arr[0]=mid;
    arr[1]=mid;

    //handle duplicates - left
    int t1 = mid;
    while(t1 >left && nums[t1]==nums[t1-1]){
        t1--;
        arr[0]=t1;
    }

    //handle duplicates - right
    int t2 = mid;
    while(t2 < right&& nums[t2]==nums[t2+1]){
        t2++;
        arr[1]=t2;
    }
    return;
}
}
```

---

# 65 Basic Calculator

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open ( and closing parentheses ), the plus + or minus sign -, non-negative integers and empty spaces. You may assume that the given expression is always valid.

Some examples: "1 + 1" = 2, "(1)" = 1, "(1-(4-5))" = 2

## 65.1 Analysis

This problem can be solved by using a stack. We keep pushing element to the stack, when ")" is met, calculate the expression up to the first "(".

## 65.2 Java Solution

---

```
public int calculate(String s) {
    // delte white spaces
    s = s.replaceAll(" ", "");

    Stack<String> stack = new Stack<String>();
    char[] arr = s.toCharArray();

    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == ' ')
            continue;

        if (arr[i] >= '0' && arr[i] <= '9') {
            sb.append(arr[i]);

            if (i == arr.length - 1) {
                stack.push(sb.toString());
            }
        } else {
            if (sb.length() > 0) {
                stack.push(sb.toString());
                sb = new StringBuilder();
            }
        }

        if (arr[i] != ')') {
            stack.push(new String(new char[] { arr[i] }));
        }
    }

    int result = 0;
    while (!stack.isEmpty()) {
        String str = stack.pop();
        if (str.charAt(0) == '+')
            result += Integer.parseInt(str.substring(1));
        else if (str.charAt(0) == '-')
            result -= Integer.parseInt(str.substring(1));
    }

    return result;
}
```

```
    } else {
        // when meet ')', pop and calculate
        ArrayList<String> t = new ArrayList<String>();
        while (!stack.isEmpty()) {
            String top = stack.pop();
            if (top.equals("(")) {
                break;
            } else {
                t.add(0, top);
            }
        }

        int temp = 0;
        if (t.size() == 1) {
            temp = Integer.valueOf(t.get(0));
        } else {
            for (int j = t.size() - 1; j > 0; j = j - 2) {
                if (t.get(j - 1).equals("-")) {
                    temp += 0 - Integer.valueOf(t.get(j));
                } else {
                    temp += Integer.valueOf(t.get(j));
                }
            }
            temp += Integer.valueOf(t.get(0));
        }
        stack.push(String.valueOf(temp));
    }
}

ArrayList<String> t = new ArrayList<String>();
while (!stack.isEmpty()) {
    String elem = stack.pop();
    t.add(0, elem);
}

int temp = 0;
for (int i = t.size() - 1; i > 0; i = i - 2) {
    if (t.get(i - 1).equals("-")) {
        temp += 0 - Integer.valueOf(t.get(i));
    } else {
        temp += Integer.valueOf(t.get(i));
    }
}
temp += Integer.valueOf(t.get(0));

return temp;
}
```

---

# 66 Group Anagrams

Given an array of strings, return all groups of strings that are anagrams.

## 66.1 Analysis

An anagram is a type of word play, the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once; for example Torchwood can be rearranged into Doctor Who.

If two strings are anagram to each other, their sorted sequence is the same.

Updated on 5/1/2016.

## 66.2 Java Solution

---

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

    HashMap<String, ArrayList<String>> map = new HashMap<String,
        ArrayList<String>>();
    for(String str: strs){
        char[] arr = str.toCharArray();
        Arrays.sort(arr);
        String ns = new String(arr);

        if(map.containsKey(ns)){
            map.get(ns).add(str);
        }else{
            ArrayList<String> al = new ArrayList<String>();
            al.add(str);
            map.put(ns, al);
        }
    }

    for(Map.Entry<String, ArrayList<String>> entry: map.entrySet()){
        Collections.sort(entry.getValue());
    }

    result.addAll(map.values());
}

return result;
}
```

---

### 66.3 Time Complexity

If average length of verbs is m and words array length is n, then the time is  $O(n*m*\log(m))$ .

# 67 Shortest Palindrome

Given a string S, you are allowed to convert it to a palindrome by adding characters in front of it. Find and return the shortest palindrome you can find by performing this transformation.

For example, given "aacecaaa", return "aaacecaaa"; given "abcd", return "dcbabcd".

## 67.1 Java Solution 1

---

```
public String shortestPalindrome(String s) {
    int i=0;
    int j=s.length()-1;

    while(j>=0){
        if(s.charAt(i)==s.charAt(j)){
            i++;
        }
        j--;
    }

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

    String suffix = s.substring(i);
    String prefix = new StringBuilder(suffix).reverse().toString();
    String mid = shortestPalindrome(s.substring(0, i));
    return prefix+mid+suffix;
}
```

---

## 67.2 Java Solution 2

We can solve this problem by using one of the methods which is used to solve the longest palindrome substring problem.

Specifically, we can start from the center and scan two sides. If read the left boundary, then the shortest palindrome is identified.

---

```
public String shortestPalindrome(String s) {
    if (s == null || s.length() <= 1)
        return s;
```

## 67 Shortest Palindrome

---

```
String result = null;

int len = s.length();
int mid = len / 2;

for (int i = mid; i >= 1; i--) {
    if (s.charAt(i) == s.charAt(i - 1)) {
        if ((result = scanFromCenter(s, i - 1, i)) != null)
            return result;
    } else {
        if ((result = scanFromCenter(s, i - 1, i - 1)) != null)
            return result;
    }
}

return result;
}

private String scanFromCenter(String s, int l, int r) {
    int i = 1;

    //scan from center to both sides
    for (; l - i >= 0; i++) {
        if (s.charAt(l - i) != s.charAt(r + i))
            break;
    }

    //if not end at the beginning of s, return null
    if (l - i >= 0)
        return null;

    StringBuilder sb = new StringBuilder(s.substring(r + i));
    sb.reverse();

    return sb.append(s).toString();
}
```

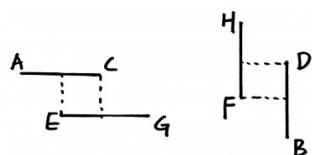
---

# 68 Rectangle Area

Find the total area covered by two rectilinear rectangles in a 2D plane. Each rectangle is defined by its bottom left corner and top right corner coordinates.

## 68.1 Analysis

This problem can be converted as a overlap internal problem. On the x-axis, there are (A,C) and (E,G); on the y-axis, there are (F,H) and (B,D). If they do not have overlap, the total area is the sum of 2 rectangle areas. If they have overlap, the total area should minus the overlap area.



## 68.2 Java Solution

---

```
public int computeArea(int A, int B, int C, int D, int E, int F, int G, int
H) {
    if(C<E || G<A )
        return (G-E)*(H-F) + (C-A)*(D-B);

    if(D<F || H<B)
        return (G-E)*(H-F) + (C-A)*(D-B);

    int right = Math.min(C,G);
    int left = Math.max(A,E);
    int top = Math.min(H,D);
    int bottom = Math.max(F,B);

    return (G-E)*(H-F) + (C-A)*(D-B) - (right-left)*(top-bottom);
}
```

---



# 69 Summary Ranges

Given a sorted integer array without duplicates, return the summary of its ranges for consecutive numbers.

For example, given [0,1,2,4,5,7], return ["0->2","4->5","7"].

## 69.1 Analysis

## 69.2 Java Solution

---

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

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

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

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

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

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

        first = nums[i];
    }
}
```

## 69 Summary Ranges

---

```
    pre = nums[i];
}

return result;
}
```

---

# 70 Increasing Triplet Subsequence

Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.

Examples: Given [1, 2, 3, 4, 5], return true.

Given [5, 4, 3, 2, 1], return false.

## 70.1 Analysis

This problem can be converted to be finding if there is a sequence such that the\_smallest\_so\_far <the\_second\_smallest\_so\_far <current. We use x, y and z to denote the 3 number respectively.

## 70.2 Java Solution

---

```
public boolean increasingTriplet(int[] nums) {
    int x = Integer.MAX_VALUE;
    int y = Integer.MAX_VALUE;

    for (int i = 0; i < nums.length; i++) {
        int z = nums[i];

        if (x >= z) {
            x = z;// update x to be a smaller value
        } else if (y >= z) {
            y = z; // update y to be a smaller value
        } else {
            return true;
        }
    }

    return false;
}
```

---



# 71 Get Target Number Using Number List and Arithmetic Operations

Given a list of numbers and a target number, write a program to determine whether the target number can be calculated by applying "+-\*/" operations to the number list? You can assume () is automatically added when necessary.

For example, given 1,2,3,4 and 21, return true. Because  $(1+2)*(3+4)=21$

## 71.1 Analysis

This is a partition problem which can be solved by using depth first search.

## 71.2 Java Solution

---

```
public static boolean isReachable(ArrayList<Integer> list, int target) {
    if (list == null || list.size() == 0)
        return false;

    int i = 0;
    int j = list.size() - 1;

    ArrayList<Integer> results = getResults(list, i, j, target);

    for (int num : results) {
        if (num == target)
            return true;
    }
}

return false;
}

public static ArrayList<Integer> getResults(ArrayList<Integer> list,
    int left, int right, int target) {
    ArrayList<Integer> result = new ArrayList<Integer>();

    if (left > right) {
        return result;
    } else if (left == right) {
        result.add(list.get(left));
    } else {
        for (int op : operators) {
            int num = calculate(list.get(left), list.get(right), op);
            result.add(num);
            result.addAll(getResults(list, left + 1, right, target));
        }
    }
}
```

## 71 Get Target Number Using Number List and Arithmetic Operations

---

```
        return result;
    }

    for (int i = left; i < right; i++) {

        ArrayList<Integer> result1 = getResults(list, left, i, target);
        ArrayList<Integer> result2 = getResults(list, i + 1, right, target);

        for (int x : result1) {
            for (int y : result2) {
                result.add(x + y);
                result.add(x - y);
                result.add(x * y);
                if (y != 0)
                    result.add(x / y);
            }
        }
    }

    return result;
}
```

---

## 72 Reverse Vowels of a String

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

### 72.1 Java Solution

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

---

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

    char[] arr = s.toCharArray();

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

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

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

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

        i++;
        j--;
    }
}
```

## 72 Reverse Vowels of a String

---

```
    }  
  
    return new String(arr);  
}
```

---

## 73 Flip Game

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive "++" into "-". The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to compute all possible states of the string after one valid move.

### 73.1 Java Solution

---

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

    if(s==null)
        return result;

    char[] arr = s.toCharArray();
    for(int i=0; i<arr.length-1; i++){
        if(arr[i]==arr[i+1] && arr[i]=='+'){
            arr[i]='-';
            arr[i+1]='-';
            result.add(new String(arr));
            arr[i]='+';
            arr[i+1]='+';
        }
    }

    return result;
}
```

---



## 74 Flip Game II

You are playing the following Flip Game with your friend: Given a string that contains only these two characters: + and -, you and your friend take turns to flip two consecutive "++" into "-". The game ends when a person can no longer make a move and therefore the other person will be the winner.

Write a function to determine if the starting player can guarantee a win.

For example, given s = "++++", return true. The starting player can guarantee a win by flipping the middle "++" to become "+--".

### 74.1 Java Solution

This problem is solved by backtracking.

---

```
public boolean canWin(String s) {
    if(s==null||s.length()==0){
        return false;
    }

    return canWinHelper(s.toCharArray());
}

public boolean canWinHelper(char[] arr){
    for(int i=0; i<arr.length-1;i++){
        if(arr[i]=='+&&arr[i+1]=='+'){
            arr[i]='-';
            arr[i+1]='-';

            boolean win = canWinHelper(arr);

            arr[i]='+';
            arr[i+1]='+';

            //if there is a flip which makes the other player lose, the first
            //play wins
            if(!win){
                return true;
            }
        }
    }

    return false;
}
```

---

## 74.2 Time Complexity

Roughly, the time is  $n \cdot n \cdot \dots \cdot n$ , which is  $O(n^n)$ . The reason is each recursion takes  $O(n)$  and there are totally  $n$  recursions.

# 75 Missing Number

Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array. For example, given nums = [0, 1, 3] return 2.

## 75.1 Java Solution 1

---

```
public int missingNumber(int[] nums) {  
    int sum=0;  
    for(int i=0; i<nums.length; i++){  
        sum+=nums[i];  
    }  
  
    int n=nums.length;  
    return n*(n+1)/2-sum;  
}
```

---

## 75.2 Java Solution 2

---

```
public int missingNumber(int[] nums) {  
  
    int miss=0;  
    for(int i=0; i<nums.length; i++){  
        miss ^= (i+1) ^ nums[i];  
    }  
  
    return miss;  
}
```

---



## 76 Valid Anagram

Given two strings s and t, write a function to determine if t is an anagram of s.

### 76.1 Java Solution 1

If the string contains only lowercase alphabets, here is a simple solution.

---

```
public boolean isAnagram(String s, String t) {  
    if(s.length()!=t.length())  
        return false;  
  
    int[] arr = new int[26];  
    for(int i=0; i<s.length(); i++){  
        char c1 = s.charAt(i);  
        arr[c1-'a']++;  
    }  
  
    for(int i=0; i<s.length(); i++){  
        char c2 = t.charAt(i);  
        if(arr[c2-'a'] == 0){  
            return false;  
        }else{  
            arr[c2-'a']--;  
        }  
    }  
  
    for(int i=0; i<26; i++){  
        if(arr[i]%2==1){  
            return false;  
        }  
    }  
  
    return true;  
}
```

---

### 76.2 Java Solution 2

If the inputs contain unicode characters, an array with length of 26 is not enough.

---

```
public boolean isAnagram(String s, String t) {  
    if(s.length()!=t.length())
```

```
    return false;

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

for(int i=0; i<s.length(); i++){
    char c1 = s.charAt(i);
    if(map.containsKey(c1)){
        map.put(c1, map.get(c1)+1);
    }else{
        map.put(c1,1);
    }
}

for(int i=0; i<t.length(); i++){
    char c2 = t.charAt(i);
    if(map.containsKey(c2)){
        if(map.get(c2)==1){
            map.remove(c2);
        }else{
            map.put(c2, map.get(c2)-1);
        }
    }else{
        return false;
    }
}

if(map.size()>0)
    return false;

return true;
}
```

---

## 77 Group Shifted Strings

Given a string, we can "shift" each of its letter to its successive letter, for example: "abc" ->"bcd". We can keep "shifting" which forms the sequence: "abc" ->"bcd" ->... ->"xyz".

Given a list of strings which contains only lowercase alphabets, group all strings that belong to the same shifting sequence, return:

```
[  
  ["abc", "bcd", "xyz"],  
  ["az", "ba"],  
  ["acef"],  
  ["a", "z"]  
]
```

### 77.1 Java Solution

```
public List<List<String>> groupStrings(String[] strings) {  
    List<List<String>> result = new ArrayList<List<String>>();  
    HashMap<String, ArrayList<String>> map  
        = new HashMap<String, ArrayList<String>>();  
  
    for(String s: strings){  
        char[] arr = s.toCharArray();  
        if(arr.length>0){  
            int diff = arr[0]-'a';  
            for(int i=0; i<arr.length; i++){  
                if(arr[i]-diff<'a'){  
                    arr[i] = (char) (arr[i]-diff+26);  
                }else{  
                    arr[i] = (char) (arr[i]-diff);  
                }  
            }  
            String ns = new String(arr);  
            if(map.containsKey(ns)){  
                map.get(ns).add(s);  
            }else{  
                ArrayList<String> al = new ArrayList<String>();  
                al.add(s);  
                map.put(ns, al);  
            }  
        }  
    }  
}
```

## 77 Group Shifted Strings

---

```
        }
    }

    for(Map.Entry<String, ArrayList<String>> entry: map.entrySet()){
        Collections.sort(entry.getValue());
    }

    result.addAll(map.values());

    return result;
}
```

---

# 78 Top K Frequent Elements

Given a non-empty array of integers, return the k most frequent elements.

## 78.1 Java Solution 1 - Using HashMap and Heap

Time is  $O(n * \log(k))$ .

---

```
class Pair{
    int num;
    int count;
    public Pair(int num, int count){
        this.num=num;
        this.count=count;
    }
}

public class Solution {
    public List<Integer> topKFrequent(int[] nums, int k) {
        //count the frequency for each element
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for(int num: nums){
            if(map.containsKey(num)){
                map.put(num, map.get(num)+1);
            }else{
                map.put(num, 1);
            }
        }

        // create a min heap
        PriorityQueue<Pair> queue = new PriorityQueue<Pair>(new
            Comparator<Pair>(){
                public int compare(Pair a, Pair b){
                    return a.count-b.count;
                }
            });
        }

        //maintain a heap of size k.
        for(Map.Entry<Integer, Integer> entry: map.entrySet()){
            Pair p = new Pair(entry.getKey(), entry.getValue());
            queue.offer(p);
            if(queue.size()>k){
                queue.poll();
            }
        }
    }
}
```

```
    }

    //get all elements from the heap
    List<Integer> result = new ArrayList<Integer>();
    while(queue.size()>0){
        result.add(queue.poll().num);
    }
    //reverse the order
    Collections.reverse(result);

    return result;
}
}
```

---

## 78.2 Java Solution 2 - Bucket Sort

Time is  $O(n)$ .

```
public List<Integer> topKFrequent(int[] nums, int k) {
    //count the frequency for each element
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    for(int num: nums){
        if(map.containsKey(num)){
            map.put(num, map.get(num)+1);
        }else{
            map.put(num, 1);
        }
    }

    //get the max frequency
    int max = 0;
    for(Map.Entry<Integer, Integer> entry: map.entrySet()){
        max = Math.max(max, entry.getValue());
    }

    //initialize an array of ArrayList. index is frequency, value is list of
    //numbers
    ArrayList<Integer>[] arr = (ArrayList<Integer>[]) new ArrayList[max+1];
    for(int i=1; i<=max; i++){
        arr[i]=new ArrayList<Integer>();
    }

    for(Map.Entry<Integer, Integer> entry: map.entrySet()){
        int count = entry.getValue();
        int number = entry.getKey();
        arr[count].add(number);
    }
}
```

```
List<Integer> result = new ArrayList<Integer>();

//add most frequent numbers to result
for(int j=max; j>=1; j--){
    if(arr[j].size()>0){
        for(int a: arr[j]){
            result.add(a);
        }
    }

    if(result.size()==k)
        break;
}

return result;
}
```

---

### 78.3 Java Solution 3 - A Regular Counter (Deprecated)

We can solve this problem by using a regular counter, and then `sort` the counter by value.

```
public class Solution {
    public List<Integer> topKFrequent(int[] nums, int k) {
        List<Integer> result = new ArrayList<Integer>();

        HashMap<Integer, Integer> counter = new HashMap<Integer, Integer>();

        for(int i: nums){
            if(counter.containsKey(i)){
                counter.put(i, counter.get(i)+1);
            }else{
                counter.put(i, 1);
            }
        }

        TreeMap<Integer, Integer> sortedMap = new TreeMap<Integer, Integer>(new
            ValueComparator(counter));
        sortedMap.putAll(counter);

        int i=0;
        for(Map.Entry<Integer, Integer> entry: sortedMap.entrySet()){
            result.add(entry.getKey());
            i++;
            if(i==k)
                break;
        }
    }
}
```

## 78 Top K Frequent Elements

---

```
        return result;
    }
}

class ValueComparator implements Comparator<Integer>{
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();

    public ValueComparator(HashMap<Integer, Integer> m){
        map.putAll(m);
    }

    public int compare(Integer i1, Integer i2){
        int diff = map.get(i2)-map.get(i1);

        if(diff==0){
            return 1;
        }else{
            return diff;
        }
    }
}
```

---

# 79 Find Peak Element

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

You may imagine that  $\text{num}[-1] = \text{num}[n] = -\infty$ . For example, in array [1, 2, 3, 1], 3 is a peak element and your function should return the index number 2.

## 79.1 Thoughts

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

## 79.2 Java Solution

---

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

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

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

        return index;
    }
}
```

---



# 80 Word Pattern

Given a pattern and a string str, find if str follows the same pattern. Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty word in str.

## 80.1 Java Solution

---

```
public boolean wordPattern(String pattern, String str) {  
    String[] arr = str.split(" ");  
  
    if(pattern.length()!=arr.length)  
        return false;  
  
    HashMap<Character, String> map = new HashMap<Character, String>();  
  
    for(int i=0; i<arr.length; i++){  
        char c = pattern.charAt(i);  
        String s = arr[i];  
  
        if(map.containsKey(c)){  
            if(!map.get(c).equals(s))  
                return false;  
  
        }else{  
            if(map.containsValue(s))  
                return false;  
  
            map.put(c, s);  
        }  
    }  
  
    return true;  
}
```

---



# 81 HIndex

Given an array of citations (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index. A scientist has index  $h$  if  $h$  of his/her  $N$  papers have at least  $h$  citations each, and the other  $N - h$  papers have no more than  $h$  citations each.

For example, given citations = [3, 0, 6, 1, 5], which means the researcher has 5 papers in total and each of them had received 3, 0, 6, 1, 5 citations respectively. Since the researcher has 3 papers with at least 3 citations each and the remaining two with no more than 3 citations each, his h-index is 3.

## 81.1 Java Solution

---

```
public int hIndex(int[] citations) {
    Arrays.sort(citations);

    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    for(int i=0; i<citations.length; i++){
        if(!map.containsKey(citations[i])){ //handle duplicates
            map.put(citations[i], citations.length-i);
        }
    }

    int result = 0;
    for(Map.Entry<Integer, Integer> entry: map.entrySet()){
        if(entry.getValue()>=entry.getKey()){
            result = Math.max(result, entry.getKey());
        }else{
            result = Math.max(result, entry.getValue());
        }
    }

    return result;
}
```

---



## 82 Palindrome Pairs

Given a list of unique words. Find all pairs of distinct indices  $(i, j)$  in the given list, so that the concatenation of the two words, i.e.  $\text{words}[i] + \text{words}[j]$  is a palindrome.

Example 1: Given words = ["bat", "tab", "cat"] Return [[0, 1], [1, 0]] The palindromes are ["battab", "tabbat"]

### 82.1 Java Solution

---

```
public List<List<Integer>> palindromePairs(String[] words) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();

    HashMap<String, Integer> map = new HashMap<String, Integer>();
    for(int i=0; i<words.length; i++){
        map.put(words[i], i);
    }

    for(int i=0; i<words.length; i++){
        String s = words[i];

        //if the word is a palindrome, get index of ""
        if(isPalindrome(s)){
            if(map.containsKey("")){
                if(map.get("")!=i){
                    ArrayList<Integer> l = new ArrayList<Integer>();
                    l.add(i);
                    l.add(map.get(""));
                    result.add(l);

                    l = new ArrayList<Integer>();
                    l.add(map.get(""));
                    l.add(i);
                    result.add(l);
                }
            }
        }

        //if the reversed word exists, it is a palindrome
        String reversed = new StringBuilder(s).reverse().toString();
        if(map.containsKey(reversed)){
            if(map.get(reversed)!=i){
                ArrayList<Integer> l = new ArrayList<Integer>();
                l.add(i);
                l.add(map.get(reversed));
                result.add(l);
            }
        }
    }
}
```

```
        if(map.get(reversed)!=i){
            ArrayList<Integer> l = new ArrayList<Integer>();
            l.add(i);
            l.add(map.get(reversed));
            result.add(l);
        }
    }

    for(int k=1; k<s.length(); k++){
        String left = s.substring(0, k);
        String right= s.substring(k);

        //if left part is palindrome, find reversed right part
        if(isPalindrome(left)){
            String reversedRight = new
                StringBuilder(right).reverse().toString();
            if(map.containsKey(reversedRight)){
                if(map.get(reversedRight)!=i){
                    ArrayList<Integer> l = new ArrayList<Integer>();
                    l.add(map.get(reversedRight));
                    l.add(i);
                    result.add(l);
                }
            }
        }

        //if right part is a palindrome, find reversed left part
        if(isPalindrome(right)){
            String reversedLeft = new
                StringBuilder(left).reverse().toString();
            if(map.containsKey(reversedLeft)){
                if(map.get(reversedLeft)!=i){

                    ArrayList<Integer> l = new ArrayList<Integer>();
                    l.add(i);
                    l.add(map.get(reversedLeft));
                    result.add(l);
                }
            }
        }
    }

    return result;
}

public boolean isPalindrome(String s){

    int i=0;
```

```
int j=s.length()-1;

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

    i++;
    j--;
}
return true;
}
```

---



## 83 One Edit Distance

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

### 83.1 Java Solution

---

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

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

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

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

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

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

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

### 83 One Edit Distance

---

```
}

if(count==1)
    return true;

return false;
}
```

---

## 84 Scramble String

Given two strings  $s_1$  and  $s_2$  of the same length, determine if  $s_2$  is a scrambled string of  $s_1$ .

### 84.1 Java Solution

---

```
public boolean isScramble(String s1, String s2) {
    if(s1.length()!=s2.length())
        return false;

    if(s1.length()==0 || s1.equals(s2))
        return true;

    char[] arr1 = s1.toCharArray();
    char[] arr2 = s2.toCharArray();
    Arrays.sort(arr1);
    Arrays.sort(arr2);
    if(!new String(arr1).equals(new String(arr2))){
        return false;
    }

    for(int i=1; i<s1.length(); i++){
        String s11 = s1.substring(0, i);
        String s12 = s1.substring(i, s1.length());
        String s21 = s2.substring(0, i);
        String s22 = s2.substring(i, s2.length());
        String s23 = s2.substring(0, s2.length()-i);
        String s24 = s2.substring(s2.length()-i, s2.length());

        if(isScramble(s11, s21) && isScramble(s12, s22))
            return true;
        if(isScramble(s11, s24) && isScramble(s12, s23))
            return true;
    }

    return false;
}
```

---



## 85 First Bad Version

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have  $n$  versions  $[1, 2, \dots, n]$  and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool `isBadVersion(version)` which will return whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

### 85.1 Java Solution 1 - Recurisve

---

```
public int firstBadVersion(int n) {
    return helper(1, n);
}

public int helper(int i, int j){
    int m = i + (j-i)/2;

    if(i>=j)
        return i;

    if(isBadVersion(m)){
        return helper(i, m);
    }else{
        return helper(m+1, j); //not bad, left --> m+1
    }
}
```

---

### 85.2 Java Solution 2 - Iterative

---

```
public int firstBadVersion(int n) {
    int i = 1, j = n;
    while (i < j) {
        int m = i + (j-i) / 2;
        if (isBadVersion(m)) {
            j = m;
        }
    }
}
```

---

## 85 First Bad Version

---

```
    } else {
        i = m+1;
    }
}

if (isBadVersion(i)) {
    return i;
}

return j;
}
```

---

# 86 Set Matrix Zeroes

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

## 86.1 Analysis

This problem should be solved in place, i.e., no other array should be used. We can use the first column and the first row to track if a row/column should be set to 0.

Since we used the first row and first column to mark the zero row/column, the original values are changed.

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

Step 1: First row contains zero = true; First column contains zero = false;

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

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

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

## 86.2 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;
}

}
```

---



## 87 Spiral Matrix

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

For example, given the following matrix:

---

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

---

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

### 87.1 Java Solution 1

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

---

```
public class Solution {  
    public ArrayList<Integer> spiralOrder(int[][] matrix) {  
        ArrayList<Integer> result = new ArrayList<Integer>();  
  
        if(matrix == null || matrix.length == 0) return result;  
  
        int m = matrix.length;  
        int n = matrix[0].length;  
  
        int x=0;  
        int y=0;  
  
        while(m>0 && n>0){  
  
            //if one row/column left, no circle can be formed  
            if(m==1){  
                for(int i=0; i<n; i++){  
                    result.add(matrix[x][y++]);  
                }  
                break;  
            }else if(n==1){  
                for(int i=0; i<m; i++){  
                    result.add(matrix[x++][y]);  
                }  
            }  
        }  
    }  
}
```

---

```
        break;
    }

    //below, process a circle

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

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

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

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

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

return result;
}
}
```

---

## 87.2 Java Solution 2

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

---

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

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

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

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

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

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

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

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

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

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

    return result;
}
```

---



## 88 Spiral Matrix II

Given an integer n, generate a square matrix filled with elements from 1 to  $n^2$  in spiral order. For example, given n = 4,

```
[  
[1, 2, 3, 4],  
[12, 13, 14, 5],  
[11, 16, 15, 6],  
[10, 9, 8, 7]  
]
```

### 88.1 Java Solution

```
public int[][] generateMatrix(int n) {  
    int total = n*n;  
    int[][] result= new int[n][n];  
  
    int x=0;  
    int y=0;  
    int step = 0;  
  
    for(int i=0;i<total;){  
        while(y+step<n){  
            i++;  
            result[x][y]=i;  
            y++;  
        }  
        y--;  
        x++;  
  
        while(x+step<n){  
            i++;  
            result[x][y]=i;  
            x++;  
        }  
        x--;  
        y--;  
  
        while(y>=0+step){  
            i++;  
        }  
    }  
}
```

## 88 Spiral Matrix II

---

```
        result[x][y]=i;
        y--;
    }
    y++;
    x--;
    step++;
}

while(x>=0+step){
    i++;
    result[x][y]=i;
    x--;
}
x++;
y++;
}

return result;
}
```

---

## 89 Search a 2D Matrix

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

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

For example, consider the following matrix:

---

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

---

Given target = 3, return true.

### 89.1 Java Solution

This is a typical problem of binary search.

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

---

```
public class Solution {  
    public boolean searchMatrix(int[][] matrix, int target) {  
        if(matrix==null || matrix.length==0 || matrix[0].length==0)  
            return false;  
  
        int m = matrix.length;  
        int n = matrix[0].length;  
  
        int start = 0;  
        int end = m*n-1;  
  
        while(start<=end){  
            int mid=(start+end)/2;  
            int midX=mid/n;  
            int midY=mid%n;  
  
            if(matrix[midX][midY]==target)  
                return true;  
        }  
    }  
}
```

## 89 Search a 2D Matrix

---

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

    return false;
}

```

---

## 90 Search a 2D Matrix II

Write an efficient algorithm that searches for a value in an  $m \times n$  matrix. This matrix has the following properties:

Integers in each row are sorted in ascending from left to right. Integers in each column are sorted in ascending from top to bottom.

For example, consider the following matrix:

---

```
[  
    [1, 4, 7, 11, 15],  
    [2, 5, 8, 12, 19],  
    [3, 6, 9, 16, 22],  
    [10, 13, 14, 17, 24],  
    [18, 21, 23, 26, 30]  
]
```

---

Given target = 5, return true.

### 90.1 Java Solution 1

In a naive approach, we can use the matrix boundary to reduce the search space. Here is a simple recursive implementation.

---

```
public boolean searchMatrix(int[][] matrix, int target) {  
    int i1=0;  
    int i2=matrix.length-1;  
    int j1=0;  
    int j2=matrix[0].length-1;  
  
    return helper(matrix, i1, i2, j1, j2, target);  
}  
  
public boolean helper(int[][] matrix, int i1, int i2, int j1, int j2, int  
target){  
  
    if(i1>i2||j1>j2)  
        return false;  
  
    for(int j=j1;j<=j2;j++){  
        if(target < matrix[i1][j])  
            return helper(matrix, i1, i2, j1, j-1, target);  
        else if(target == matrix[i1][j])  
            return true;  
    }  
}
```

---

```
        }
    }

    for(int i=i1;i<=i2;i++){
        if(target < matrix[i][j1]){
            return helper(matrix, i1, i-1, j1, j2, target);
        }else if(target == matrix[i][j1]){
            return true;
        }
    }

    for(int j=j1;j<=j2;j++){
        if(target > matrix[i2][j]){
            return helper(matrix, i1, i2, j+1, j2, target);
        }else if(target == matrix[i2][j]){
            return true;
        }
    }

    for(int i=i1;i<=i2;i++){
        if(target > matrix[i][j2]){
            return helper(matrix, i1, i+1, j1, j2, target);
        }else if(target == matrix[i][j2]){
            return true;
        }
    }

    return false;
}
```

---

## 90.2 Java Solution 2

Time Complexity: O(m + n)

```
public boolean searchMatrix(int[][][] matrix, int target) {
    int m=matrix.length-1;
    int n=matrix[0].length-1;

    int i=m;
    int j=0;

    while(i>=0 && j<=n){
        if(target < matrix[i][j]){
            i--;
        }else if(target > matrix[i][j]){
            j++;
        }else{
            return true;
        }
    }
}
```

```
        }  
    }  
  
    return false;  
}
```

---



# 91 Rotate Image

You are given an  $n \times n$  2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up: Could you do this in-place?

## 91.1 Naive Solution

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

---

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

        int m = matrix.length;

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

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

        matrix = result;
    }
}
```

---

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

---

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

        int m = matrix.length;
```

## 91 Rotate Image

---

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

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

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

---

## 91.2 In-place Solution

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

---

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

---

## 92 Valid Sudoku

Determine if a Sudoku is valid. The Sudoku board could be partially filled, where empty cells are filled with the character '.'.

5	3			7				
6			1	9	5			
	9	8				6		
8				6				3
4			8		3			1
7				2				6
	6				2	8		
			4	1	9			5
				8		7	9	

### 92.1 Java Solution

---

```
public boolean isValidSudoku(char[][] board) {
    if (board == null || board.length != 9 || board[0].length != 9)
        return false;
    // check each column
    for (int i = 0; i < 9; i++) {
        boolean[] m = new boolean[9];
        for (int j = 0; j < 9; j++) {
            if (board[i][j] != '.') {
                if (m[(int) (board[i][j] - '1')]) {
                    return false;
                }
                m[(int) (board[i][j] - '1')] = true;
            }
        }
    }

    //check each row
    for (int j = 0; j < 9; j++) {
        boolean[] m = new boolean[9];
        for (int i = 0; i < 9; i++) {
```

```
if (board[i][j] != '.') {
    if (m[(int) (board[i][j] - '1')]) {
        return false;
    }
    m[(int) (board[i][j] - '1')] = true;
}
}

//check each 3*3 matrix
for (int block = 0; block < 9; block++) {
    boolean[] m = new boolean[9];
    for (int i = block / 3 * 3; i < block / 3 * 3 + 3; i++) {
        for (int j = block % 3 * 3; j < block % 3 * 3 + 3; j++) {
            if (board[i][j] != '.') {
                if (m[(int) (board[i][j] - '1')]) {
                    return false;
                }
                m[(int) (board[i][j] - '1')] = true;
            }
        }
    }
}

return true;
}
```

---

## 93 Minimum Path Sum

Given a  $m \times n$  grid filled with non-negative numbers, find a path from top left to bottom right which minimizes the sum of all numbers along its path.

### 93.1 Java Solution 1: Depth-First Search

A native solution would be depth-first search. It's time is too expensive and fails the online judgement.

---

```
public int minPathSum(int[][][] grid) {
    return dfs(0,0,grid);
}

public int dfs(int i, int j, int[][][] grid){
    if(i==grid.length-1 && j==grid[0].length-1){
        return grid[i][j];
    }

    if(i<grid.length-1 && j<grid[0].length-1){
        int r1 = grid[i][j] + dfs(i+1, j, grid);
        int r2 = grid[i][j] + dfs(i, j+1, grid);
        return Math.min(r1,r2);
    }

    if(i<grid.length-1){
        return grid[i][j] + dfs(i+1, j, grid);
    }

    if(j<grid[0].length-1){
        return grid[i][j] + dfs(i, j+1, grid);
    }

    return 0;
}
```

---

### 93.2 Java Solution 2: Dynamic Programming

---

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

### 93 Minimum Path Sum

---

```
int m = grid.length;
int n = grid[0].length;

int[][] dp = new int[m][n];
dp[0][0] = grid[0][0];

// initialize top row
for(int i=1; i<n; i++){
    dp[0][i] = dp[0][i-1] + grid[0][i];
}

// initialize left column
for(int j=1; j<m; j++){
    dp[j][0] = dp[j-1][0] + grid[j][0];
}

// fill up the dp table
for(int i=1; i<m; i++){
    for(int j=1; j<n; j++){
        if(dp[i-1][j] > dp[i][j-1]){
            dp[i][j] = dp[i][j-1] + grid[i][j];
        }else{
            dp[i][j] = dp[i-1][j] + grid[i][j];
        }
    }
}

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

---

## 94 Unique Paths

A robot is located at the top-left corner of a  $m \times n$  grid. It can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid.

How many possible unique paths are there?

### 94.1 Java Solution 1 - DFS

A depth-first search solution is pretty straight-forward. However, the time of this solution is too expensive, and it didn't pass the online judge.

---

```
public int uniquePaths(int m, int n) {
    return dfs(0,0,m,n);
}

public int dfs(int i, int j, int m, int n){
    if(i==m-1 && j==n-1){
        return 1;
    }

    if(i<m-1 && j<n-1){
        return dfs(i+1,j,m,n) + dfs(i,j+1,m,n);
    }

    if(i<m-1){
        return dfs(i+1,j,m,n);
    }

    if(j<n-1){
        return dfs(i,j+1,m,n);
    }

    return 0;
}
```

---

### 94.2 Java Solution 2 - Dynamic Programming

---

```
public int uniquePaths(int m, int n) {
    if(m==0 || n==0) return 0;
    if(m==1 || n==1) return 1;
```

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

//left column
for(int i=0; i<m; i++){
    dp[i][0] = 1;
}

//top row
for(int j=0; j<n; j++){
    dp[0][j] = 1;
}

//fill up the dp table
for(int i=1; i<m; i++){
    for(int j=1; j<n; j++){
        dp[i][j] = dp[i-1][j] + dp[i][j-1];
    }
}

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

---

## 95 Unique Paths II

Follow up for "Unique Paths":

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and empty space is marked as 1 and 0 respectively in the grid. For example, there is one obstacle in the middle of a  $3 \times 3$  grid as illustrated below,

```
[  
  [0,0,0],  
  [0,1,0],  
  [0,0,0]  
]
```

the total number of unique paths is 2.

### 95.1 Java Solution

```
public int uniquePathsWithObstacles(int[][] obstacleGrid) {  
    if(obstacleGrid==null||obstacleGrid.length==0)  
        return 0;  
  
    int m = obstacleGrid.length;  
    int n = obstacleGrid[0].length;  
  
    if(obstacleGrid[0][0]==1||obstacleGrid[m-1][n-1]==1)  
        return 0;  
  
    int[][] dp = new int[m][n];  
    dp[0][0]=1;  
  
    //left column  
    for(int i=1; i<m; i++){  
        if(obstacleGrid[i][0]==1){  
            dp[i][0] = 0;  
        }else{  
            dp[i][0] = dp[i-1][0];  
        }  
    }  
  
    //top row
```

## 95 Unique Paths II

---

```
for(int i=1; i<n; i++){
    if(obstacleGrid[0][i]==1){
        dp[0][i] = 0;
    }else{
        dp[0][i] = dp[0][i-1];
    }
}

//fill up cells inside
for(int i=1; i<m; i++){
    for(int j=1; j<n; j++){
        if(obstacleGrid[i][j]==1){
            dp[i][j]=0;
        }else{
            dp[i][j]=dp[i-1][j]+dp[i][j-1];
        }
    }
}

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

---

## 96 Number of Islands

Given a 2-d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

---

```
11110
11010
11000
00000
```

---

Answer: 1

Example 2:

---

```
11000
11000
00100
00011
```

---

Answer: 3

### 96.1 Java Solution

The basic idea of the following solution is merging adjacent lands, and the merging should be done recursively.

---

```
public int numIslands(char[][] grid) {
    if(grid==null||grid.length==0||grid[0].length==0)
        return 0;

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

    int count=0;
    for(int i=0;i<m; i++){
        for(int j=0; j<n; j++){
            if(grid[i][j]=='1'){
                count++;
                merge(grid, i, j);
            }
        }
    }
}
```

```
    return count;
}

public void merge(char[][] grid, int i, int j){
    if(i<0||j<0||i>=grid.length||j>=grid[0].length)
        return;

    if(grid[i][j]=='1'){
        grid[i][j]='0';

        merge(grid, i-1,j);
        merge(grid, i+1,j);
        merge(grid, i,j-1);
        merge(grid, i,j+1);
    }
}
```

---

Check out Number of Island II.

## 97 Number of Islands II

A 2d grid map of m rows and n columns is initially filled with water. We may perform an addLand operation which turns the water at position (row, col) into a land. Given a list of positions to operate, count the number of islands after each addLand operation. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

### 97.1 Java Solution

Use an array to track the parent node for each cell.

```
public List<Integer> numIslands2(int m, int n, int[][][] positions) {
    int[] rootArray = new int[m*n];
    Arrays.fill(rootArray,-1);

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

    int[][][] directions = {{-1,0},{0,1},{1,0},{0,-1}};
    int count=0;

    for(int k=0; k<positions.length; k++){
        count++;

        int[] p = positions[k];
        int index = p[0]*n+p[1];
        rootArray[index]=index;//set root to be itself for each node

        for(int r=0;r<4;r++){
            int i=p[0]+directions[r][0];
            int j=p[1]+directions[r][1];

            if(i>=0&&j>=0&&i<m&&j<n&&rootArray[i*n+j]!=-1){
                //get neighbor's root
                int thisRoot = getRoot(rootArray, i*n+j);
                if(thisRoot!=index){
                    rootArray[thisRoot]=index;//set previous root's root
                    count--;
                }
            }
        }
    }
}
```

## 97 Number of Islands II

---

```
        result.add(count);
    }

    return result;
}

public int getRoot(int[] arr, int i){
    while(i!=arr[i]){
        i=arr[arr[i]];
    }
    return i;
}
```

---

## 98 Surrounded Regions

Given a 2D board containing 'X' and 'O', capture all regions surrounded by 'X'. A region is captured by flipping all 'O's into 'X's in that surrounded region.

For example,

---

```
X X X X
X O O X
X X O X
X O X X
```

---

After running your function, the board should be:

---

```
X X X X
X X X X
X X X X
X O X X
```

---

### 98.1 Analysis

This problem is similar to [Number of Islands](#). In this problem, only the cells on the boarders can not be surrounded. So we can first merge those O's on the boarders like in [Number of Islands](#) and replace O's with '#', and then scan the board and replace all O's left (if any).

### 98.2 Depth-first Search

---

```
public void solve(char[][] board) {
    if(board == null || board.length==0)
        return;

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

    //merge O's on left & right boarder
    for(int i=0;i<m;i++){
        if(board[i][0] == '0'){
            merge(board, i, 0);
        }
    }

    if(board[i][n-1] == '0'){

    }
}
```

```
        merge(board, i, n-1);
    }
}

//merge 0's on top & bottom border
for(int j=0; j<n; j++){
    if(board[0][j] == '0'){
        merge(board, 0, j);
    }

    if(board[m-1][j] == '0'){
        merge(board, m-1, j);
    }
}

//process the board
for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
        if(board[i][j] == '0'){
            board[i][j] = 'X';
        }else if(board[i][j] == '#'){
            board[i][j] = '0';
        }
    }
}

public void merge(char[][] board, int i, int j){
    if(i<0 || i>=board.length || j<0 || j>=board[0].length)
        return;

    if(board[i][j] != '0')
        return;

    board[i][j] = '#';

    merge(board, i-1, j);
    merge(board, i+1, j);
    merge(board, i, j-1);
    merge(board, i, j+1);
}
```

---

This solution causes `java.lang.StackOverflowError`, because for a large board, too many method calls are pushed to the stack and causes the overflow.

### 98.3 Breath-first Search

Instead we use a queue to do breath-first search.

---

```

public class Solution {
    // use a queue to do BFS
    private Queue<Integer> queue = new LinkedList<Integer>();

    public void solve(char[][] board) {
        if (board == null || board.length == 0)
            return;

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

        // merge 0's on left & right border
        for (int i = 0; i < m; i++) {
            if (board[i][0] == '0') {
                bfs(board, i, 0);
            }

            if (board[i][n - 1] == '0') {
                bfs(board, i, n - 1);
            }
        }

        // merge 0's on top & bottom border
        for (int j = 0; j < n; j++) {
            if (board[0][j] == '0') {
                bfs(board, 0, j);
            }

            if (board[m - 1][j] == '0') {
                bfs(board, m - 1, j);
            }
        }

        // process the board
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i][j] == '0') {
                    board[i][j] = 'X';
                } else if (board[i][j] == '#') {
                    board[i][j] = '0';
                }
            }
        }
    }

    private void bfs(char[][] board, int i, int j) {
        int n = board[0].length;

        // fill current first and then its neighbors
        fillCell(board, i, j);
    }
}

```

```
while (!queue.isEmpty()) {
    int cur = queue.poll();
    int x = cur / n;
    int y = cur % n;

    fillCell(board, x - 1, y);
    fillCell(board, x + 1, y);
    fillCell(board, x, y - 1);
    fillCell(board, x, y + 1);
}
}

private void fillCell(char[][] board, int i, int j) {
    int m = board.length;
    int n = board[0].length;
    if (i < 0 || i >= m || j < 0 || j >= n || board[i][j] != '0')
        return;

    // add current cell to queue & then process its neighbors in bfs
    queue.offer(i * n + j);
    board[i][j] = '#';
}
}
```

---

# 99 Maximal Rectangle

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing all ones and return its area.

## 99.1 Analysis

This problem can be converted to the "Largest Rectangle in Histogram" problem.

## 99.2 Java Solution

---

```
public int maximalRectangle(char[][] matrix) {
    int m = matrix.length;
    int n = m == 0 ? 0 : matrix[0].length;
    int[][] height = new int[m][n + 1];

    int maxArea = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (matrix[i][j] == '0') {
                height[i][j] = 0;
            } else {
                height[i][j] = i == 0 ? 1 : height[i - 1][j] + 1;
            }
        }
    }

    for (int i = 0; i < m; i++) {
        int area = maxAreaInHist(height[i]);
        if (area > maxArea) {
            maxArea = area;
        }
    }

    return maxArea;
}

private int maxAreaInHist(int[] height) {
    Stack<Integer> stack = new Stack<Integer>();

    int i = 0;
    int max = 0;
```

## 99 Maximal Rectangle

---

```
while (i < height.length) {  
    if (stack.isEmpty() || height[stack.peek()] <= height[i]) {  
        stack.push(i++);  
    } else {  
        int t = stack.pop();  
        max = Math.max(max, height[t]  
            * (stack.isEmpty() ? i : i - stack.peek() - 1));  
    }  
}  
  
return max;  
}
```

---

# 100 Maximal Square

Given a 2D binary matrix filled with 0's and 1's, find the largest square containing all 1's and return its area.

For example, given the following matrix:

---

```
1101
1101
1111
```

---

Return 4.

## 100.1 Analysis

This problem can be solved by dynamic programming. The changing condition is:  $t[i][j] = \min(t[i][j-1], t[i-1][j], t[i-1][j-1]) + 1$ . It means the square formed before this point.

## 100.2 Java Solution

---

```
public int maximalSquare(char[][] matrix) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0)
        return 0;

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

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

    //top row
    for (int i = 0; i < m; i++) {
        t[i][0] = Character.getNumericValue(matrix[i][0]);
    }

    //left column
    for (int j = 0; j < n; j++) {
        t[0][j] = Character.getNumericValue(matrix[0][j]);
    }

    //cells inside
    for (int i = 1; i < m; i++) {
```

## 100 Maximal Square

---

```
for (int j = 1; j < n; j++) {
    if (matrix[i][j] == '1') {
        int min = Math.min(t[i - 1][j], t[i - 1][j - 1]);
        min = Math.min(min, t[i][j - 1]);
        t[i][j] = min + 1;
    } else {
        t[i][j] = 0;
    }
}

int max = 0;
//get maximal length
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (t[i][j] > max) {
            max = t[i][j];
        }
    }
}

return max * max;
}
```

---

# 101 Word Search

Given a 2D board and a word, find if the word exists in the grid.

The word can be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

For example, given board =

---

```
[  
    ["ABCE"],  
    ["SFCS"],  
    ["ADEE"]  
]
```

---

word = "ABCED", ->returns true, word = "SEE", ->returns true, word = "ABCB", ->returns false.

## 101.1 Analysis

This problem can be solve by using a typical DFS method.

## 101.2 Java Solution

---

```
public boolean exist(char[][] board, String word) {  
    int m = board.length;  
    int n = board[0].length;  
  
    boolean result = false;  
    for(int i=0; i<m; i++){  
        for(int j=0; j<n; j++){  
            if(dfs(board,word,i,j,0)){  
                result = true;  
            }  
        }  
    }  
  
    return result;  
}  
  
public boolean dfs(char[][] board, String word, int i, int j, int k){  
    int m = board.length;
```

## 101 Word Search

---

```
int n = board[0].length;

if(i<0 || j<0 || i>=m || j>=n){
    return false;
}

if(board[i][j] == word.charAt(k)){
    char temp = board[i][j];
    board[i][j]='#';
    if(k==word.length()-1){
        return true;
    }else if(dfs(board, word, i-1, j, k+1)
    ||dfs(board, word, i+1, j, k+1)
    ||dfs(board, word, i, j-1, k+1)
    ||dfs(board, word, i, j+1, k+1)){
        return true;
    }
    board[i][j]=temp;
}

return false;
}
```

---

## 102 Word Search II

Given a 2D board and a list of words from the dictionary, find all words in the board.

Each word must be constructed from letters of sequentially adjacent cell, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

For example, given words = ["oath","pea","eat","rain"] and board =

```
[  
  ['o','a','a','n'],  
  ['e','t','a','e'],  
  ['i','h','k','r'],  
  ['i','f','l','v']  
]
```

Return ["eat","oath"].

### 102.1 Java Solution 1

Similar to [Word Search](#), this problem can be solved by DFS. However, this solution exceeds time limit.

```
public List<String> findWords(char[][] board, String[] words) {  
    ArrayList<String> result = new ArrayList<String>();  
  
    int m = board.length;  
    int n = board[0].length;  
  
    for (String word : words) {  
        boolean flag = false;  
        for (int i = 0; i < m; i++) {  
            for (int j = 0; j < n; j++) {  
                char[][] newBoard = new char[m][n];  
                for (int x = 0; x < m; x++)  
                    for (int y = 0; y < n; y++)  
                        newBoard[x][y] = board[x][y];  
  
                if (dfs(newBoard, word, i, j, 0)) {  
                    flag = true;  
                }  
            }  
        }  
        if (flag) {
```

```
        result.add(word);
    }
}

return result;
}

public boolean dfs(char[][] board, String word, int i, int j, int k) {
    int m = board.length;
    int n = board[0].length;

    if (i < 0 || j < 0 || i >= m || j >= n || k > word.length() - 1) {
        return false;
    }

    if (board[i][j] == word.charAt(k)) {
        char temp = board[i][j];
        board[i][j] = '#';

        if (k == word.length() - 1) {
            return true;
        } else if (dfs(board, word, i - 1, j, k + 1)
            || dfs(board, word, i + 1, j, k + 1)
            || dfs(board, word, i, j - 1, k + 1)
            || dfs(board, word, i, j + 1, k + 1)) {
            board[i][j] = temp;
            return true;
        }
    } else {
        return false;
    }

    return false;
}
```

---

## 102.2 Java Solution 2 - Trie

If the current candidate does not exist in all words' prefix, we can stop backtracking immediately. This can be done by using a trie structure.

---

```
public class Solution {
    Set<String> result = new HashSet<String>();

    public List<String> findWords(char[][] board, String[] words) {
        //HashSet<String> result = new HashSet<String>();

        Trie trie = new Trie();
```

```

for(String word: words){
    trie.insert(word);
}

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

boolean[][] visited = new boolean[m][n];

for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
        dfs(board, visited, "", i, j, trie);
    }
}

return new ArrayList<String>(result);
}

public void dfs(char[][] board, boolean[][] visited, String str, int i,
    int j, Trie trie){
    int m=board.length;
    int n=board[0].length;

    if(i<0 || j<0||i>=m||j>=n){
        return;
    }

    if(visited[i][j])
        return;

    str = str + board[i][j];

    if(!trie.startsWith(str))
        return;

    if(trie.search(str)){
        result.add(str);
    }

    visited[i][j]=true;
    dfs(board, visited, str, i-1, j, trie);
    dfs(board, visited, str, i+1, j, trie);
    dfs(board, visited, str, i, j-1, trie);
    dfs(board, visited, str, i, j+1, trie);
    visited[i][j]=false;
}
}

```

---

//Trie Node

```
class TrieNode{
    public TrieNode[] children = new TrieNode[26];
    public String item = "";
}

//Trie
class Trie{
    public TrieNode root = new TrieNode();

    public void insert(String word){
        TrieNode node = root;
        for(char c: word.toCharArray()){
            if(node.children[c-'a']==null){
                node.children[c-'a']= new TrieNode();
            }
            node = node.children[c-'a'];
        }
        node.item = word;
    }

    public boolean search(String word){
        TrieNode node = root;
        for(char c: word.toCharArray()){
            if(node.children[c-'a']==null)
                return false;
            node = node.children[c-'a'];
        }
        if(node.item.equals(word)){
            return true;
        }else{
            return false;
        }
    }

    public boolean startsWith(String prefix){
        TrieNode node = root;
        for(char c: prefix.toCharArray()){
            if(node.children[c-'a']==null)
                return false;
            node = node.children[c-'a'];
        }
        return true;
    }
}
```

---

# 103 Integer Break

Given a positive integer  $n$ , break it into the sum of at least two positive integers and maximize the product of those integers. Return the maximum product you can get.

For example, given  $n = 2$ , return  $1$  ( $2 = 1 + 1$ ); given  $n = 10$ , return  $36$  ( $10 = 3 + 3 + 4$ ).

## 103.1 Java Solution 1 - Dynamic Programming

Let  $dp[i]$  to be the max production value for breaking the number  $i$ . Since  $dp[i+j]$  can be  $i*j$ ,  $dp[i+j] = \max(\max(dp[i], i) * \max(dp[j], j)), dp[i+j]$ .

---

```
public int integerBreak(int n) {
    int[] dp = new int[n+1];

    for(int i=1; i<n; i++){
        for(int j=1; j<i+1; j++){
            if(i+j<=n){
                dp[i+j]=Math.max(Math.max(dp[i],i)*Math.max(dp[j],j), dp[i+j]);
            }
        }
    }

    return dp[n];
}
```

---

## 103.2 Java Solution 2 - Using Regularities

If we see the breaking result for some numbers, we can see repeated pattern like the following:

---

```
2 -> 1*1
3 -> 1*2
4 -> 2*2
5 -> 3*2
6 -> 3*3
7 -> 3*4
8 -> 3*3*2
9 -> 3*3*3
10 -> 3*3*4
11 -> 3*3*3*2
```

---

## 103 Integer Break

---

We only need to find how many 3's we can get when  $n > 4$ . If  $n$

```
public int integerBreak(int n) {  
  
    if(n==2) return 1;  
    if(n==3) return 2;  
    if(n==4) return 4;  
  
    int result=1;  
    if(n%3==0){  
        int m = n/3;  
        result = (int) Math.pow(3, m);  
    }else if(n%3==2){  
        int m=n/3;  
        result = (int) Math.pow(3, m) * 2;  
    }else if(n%3==1){  
        int m=(n-4)/3;  
        result = (int) Math.pow(3, m) *4;  
    }  
  
    return result;  
}
```

---

# 104 Range Sum Query 2D Immutable

Given a 2D matrix matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).

## 104.1 Analysis

Since the assumption is that there are many calls to sumRegion method, we should use some extra space to store the intermediate results. Here we define an array sum[][] which stores the sum value from (0,0) to the current cell.

## 104.2 Java Solution

---

```
public class NumMatrix {
    int [][] sum;

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

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

        for(int i=0; i<m; i++){
            int sumRow=0;
            for(int j=0; j<n; j++){
                if(i==0){
                    sumRow += matrix[i][j];
                    sum[i][j]=sumRow;
                }else{
                    sumRow += matrix[i][j];
                    sum[i][j]=sumRow+sum[i-1][j];
                }
            }
        }
    }

    public int sumRegion(int row1, int col1, int row2, int col2) {
        if(this.sum==null)
```

```
        return 0;

    int topRightX = row1;
    int topRightY = col2;

    int bottomLeftX=row2;
    int bottomLeftY= col1;

    int result=0;

    if(row1==0 && col1==0){
        result = sum[row2][col2];
    }else if(row1==0){
        result = sum[row2][col2]
            -sum[bottomLeftX][bottomLeftY-1];
    }else if(col1==0){
        result = sum[row2][col2]
            -sum[topRightX-1][topRightY];
    }else{
        result = sum[row2][col2]
            -sum[topRightX-1][topRightY]
            -sum[bottomLeftX][bottomLeftY-1]
            +sum[row1-1][col1-1];
    }

    return result;
}
}
```

---

# 105 Longest Increasing Path in a Matrix

Given an integer matrix, find the length of the longest increasing path.

From each cell, you can either move to four directions: left, right, up or down. You may NOT move diagonally or move outside of the boundary

## 105.1 Java Solution 1 - DFS

This solution is over time limit.

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

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

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

        return longest;
    }

    public void helper(int[][] matrix, int i, int j, int len){

        if(i-1>=0 && matrix[i-1][j]>matrix[i][j]){
            longest = Math.max(longest, len+1);
            helper(matrix, i-1, j, len+1);
        }

        if(i+1<matrix.length && matrix[i+1][j]>matrix[i][j]){
            longest = Math.max(longest, len+1);
            helper(matrix, i+1, j, len+1);
        }

        if(j-1>=0 && matrix[i][j-1]>matrix[i][j]){
            longest = Math.max(longest, len+1);
            helper(matrix, i, j-1, len+1);
        }
    }
}
```

```
        if(j+1<matrix[0].length && matrix[i][j+1]>matrix[i][j]){
            longest = Math.max(longest, len+1);
            helper(matrix, i, j+1, len+1);
        }

    }
}
```

---

## 105.2 Java Solution - Optimized

```
public class Solution {
    int[] dx = {-1, 1, 0, 0};
    int[] dy = {0, 0, -1, 1};

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

        int[][] mem = new int[matrix.length][matrix[0].length];
        int longest=0;

        for(int i=0; i<matrix.length; i++){
            for(int j=0; j<matrix[0].length; j++){
                longest = Math.max(longest, dfs(matrix, i, j, mem));
            }
        }

        return longest;
    }

    public int dfs(int[][][] matrix, int i, int j, int[][] mem){
        if(mem[i][j]!=0)
            return mem[i][j];

        for(int m=0; m<4; m++){
            int x = i+dx[m];
            int y = j+dy[m];

            if(x>=0&&y>=0&&x<matrix.length&&y<matrix[0].length&&matrix[x][y]>matrix[i][j]){
                mem[i][j]=Math.max(mem[i][j], dfs(matrix, x, y, mem));
            }
        }

        return ++mem[i][j];
    }
}
```

---

# 106 Shortest Distance from All Buildings

You want to build a house on an empty land which reaches all buildings in the shortest amount of distance. You can only move up, down, left and right. You are given a 2D grid of values 0, 1 or 2, where:

Each 0 marks an empty land which you can pass by freely. Each 1 marks a building which you cannot pass through. Each 2 marks an obstacle which you cannot pass through.

For example, given three buildings at (0,0), (0,4), (2,2), and an obstacle at (0,2). The point (1,2) is an ideal empty land to build a house, as the total travel distance of  $3+3+1=7$  is minimal. So return 7.

## 106.1 Java Solution

This problem can be solved by BFS. We define one matrix for tracking the distance from each building, and another matrix for tracking the number of buildings which can be reached.

---

```
public class Solution {

    int[][] numReach;
    int[][] distance;

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

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

        numReach = new int[m][n];
        distance = new int[m][n];

        int numBuilding = 0;
        for(int i=0; i<m; i++){
            for(int j=0; j<n; j++){
                if(grid[i][j]==1){
                    boolean[][] visited = new boolean[m][n];
                    LinkedList<Integer> queue = new LinkedList<Integer>();
                    dfs(grid, i, j, i, j, 0, visited, queue);
                }
            }
        }
    }

    void dfs(int[][] grid, int i, int j, int x, int y, int dist, boolean[][] visited, LinkedList<Integer> queue) {
        if(i<0 || j<0 || i>=grid.length || j>=grid[0].length || grid[i][j]==2 || visited[i][j])
            return;

        visited[i][j] = true;
        queue.add(4*x + 100*y + dist);

        dfs(grid, i+1, j, x+1, y, dist+1, visited, queue);
        dfs(grid, i-1, j, x-1, y, dist+1, visited, queue);
        dfs(grid, i, j+1, x, y+1, dist+1, visited, queue);
        dfs(grid, i, j-1, x, y-1, dist+1, visited, queue);
    }
}
```

```
        numBuilding++;
    }
}

int result=Integer.MAX_VALUE;
for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
        if(grid[i][j] == 0 && numReach[i][j]==numBuilding){
            result = Math.min(result, distance[i][j]);
        }
    }
}

return result == Integer.MAX_VALUE ? -1 : result;
}

public void dfs(int[][] grid, int ox, int oy, int i, int j,
                int distanceSoFar, boolean[][] visited, LinkedList<Integer>
                queue){

    visit(grid, i, j, i, j, distanceSoFar, visited, queue);
    int n = grid[0].length;

    while(!queue.isEmpty()){
        int size = queue.size();
        distanceSoFar++;

        for(int k=0; k<size; k++){
            int top = queue.poll();
            i=top/n;
            j=top%n;

            visit(grid, ox, oy, i-1, j, distanceSoFar, visited, queue);
            visit(grid, ox, oy, i+1, j, distanceSoFar, visited, queue);
            visit(grid, ox, oy, i, j-1, distanceSoFar, visited, queue);
            visit(grid, ox, oy, i, j+1, distanceSoFar, visited, queue);
        }
    }
}

public void visit(int[][] grid, int ox, int oy, int i, int j, int
                  distanceSoFar, boolean[][] visited, LinkedList<Integer> queue){
    int m = grid.length;
    int n = grid[0].length;

    if(i<0 || i>=m || j<0 || j>=n || visited[i][j])
        return;
}
```

```
if((i!=ox || j!=oy) && grid[i][j]!=0){
    return;
}

visited[i][j]=true;
numReach[i][j]++;
distance[i][j]+= distanceSoFar;
queue.offer(i*n+j);
}
}
```

---



# 107 Game of Life

Given a board with  $m$  by  $n$  cells, each cell has an initial state live (1) or dead (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules:

Any live cell with fewer than two live neighbors dies, as if caused by under-population.  
Any live cell with two or three live neighbors lives on to the next generation. Any live cell with more than three live neighbors dies, as if by over-population.. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Write a function to compute the next state (after one update) of the board given its current state.

## 107.1 Java Solution

Because we need to solve the problem in place, we can use the higher bit to record the next state. And at the end, shift right a bit to get the next state for each cell.

---

```
public void gameOfLife(int[][][] board) {
    int m = board.length;
    int n = board[0].length;

    int[] dx = {-1, -1, 0, 1, 1, 1, 0, -1};
    int[] dy = {0, 1, 1, 1, 0, -1, -1, -1};
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            int count = 0;
            for(int k=0; k<8; k++){
                int x = i+dx[k];
                int y = j+dy[k];
                count += getNeighbor(board, x, y);
            }
            //Any live cell with fewer than two live neighbors dies
            if(count<2 && board[i][j]==1){
                board[i][j] &= 1; // this line can be ignored, since next state
                is 0 by default
            }
            //Any dead cell with exactly three live neighbors becomes a live cell
            if(count==3 && board[i][j]==0){
                board[i][j] |= 2; // e.g., '01' & '10'='11'
            }
            // any live cells with 2 or 3 neighbors lives on to the next
            // generation
        }
    }
}
```

```
        if(count>=2 && count<=3 && board[i][j]==1){
            board[i][j] |= 2;
        }
        //Any live cell with more than three live neighbors dies
        if(count>3 && board[i][j]==1){
            board[i][j] &= 1; // this line can be ignored, since next state
                               // is 0 by default
        }
    }

    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            board[i][j] >= 1;
        }
    }
}

public int getNeighbor(int[][] board, int x, int y){
    int m = board.length;
    int n = board[0].length;
    if(x<0 || x>=m || y<0 || y>=n)
        return 0;
    return board[x][y]&1;
}
```

---

# 108 Implement a Stack Using an Array in Java

This post shows how to implement a stack by using an array.

The requirements of the stack are: 1) the stack has a constructor which accept a number to initialize its size, 2) the stack can hold any type of elements, 3) the stack has a push() and a pop() method.

I remember there is a similar example in the "Effective Java" book written by Joshua Bloch, but not sure how the example is used. So I just write one and then read the book, and see if I miss anything.

## 108.1 A Simple Stack Implementation

---

```
public class Stack<E> {
    private E[] arr = null;
    private int CAP;
    private int top = -1;
    private int size = 0;

    @SuppressWarnings("unchecked")
    public Stack(int cap) {
        this.CAP = cap;
        this.arr = (E[]) new Object[cap];
    }

    public E pop() {
        if(this.size == 0){
            return null;
        }

        this.size--;
        E result = this.arr[top];
        this.arr[top] = null;//prevent memory leaking
        this.top--;

        return result;
    }

    public boolean push(E e) {
        if (!isFull())
            return false;
```

```
        this.size++;
        this.arr[++top] = e;
        return false;
    }

    public boolean isFull() {
        if (this.size == this.CAP)
            return false;
        return true;
    }

    public String toString() {
        if(this.size==0){
            return null;
        }

        StringBuilder sb = new StringBuilder();
        for(int i=0; i<this.size; i++){
            sb.append(this.arr[i] + ", ");
        }

        sb.setLength(sb.length()-2);
        return sb.toString();
    }

    public static void main(String[] args) {

        Stack<String> stack = new Stack<String>(11);
        stack.push("hello");
        stack.push("world");

        System.out.println(stack);

        stack.pop();
        System.out.println(stack);

        stack.pop();
        System.out.println(stack);
    }
}
```

---

Output:

---

```
hello, world
hello
null
```

---

## 108.2 Information from "Effective Java"

It turns out I don't need to improve anything. There are some naming differences but overall my method is ok.

This example occurs twice in "Effective Java". In the first place, the stack example is used to illustrate memory leak. In the second place, the example is used to illustrate when we can suppress unchecked warnings.

Do you wonder how to implement a queue by using an array?



# 109 Add Two Numbers

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

## 109.1 Java Solution

---

```
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;
    }
}
```

---

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

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



# 110 Reorder List

Given a singly linked list L:  $L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$ , reorder it to:  $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \dots$

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

## 110.1 Analysis

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

## 110.2 Java Solution

This problem can be solved by doing the following:

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

The following code is a complete runnable class with testing.

---

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

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

public class ReorderList {

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

## 110 Reorder List

---

```
n3.next = n4;

printList(n1);

reorderList(n1);

printList(n1);
}

public static void reorderList(ListNode head) {

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

        ListNode slow = head;
        ListNode fast = head;

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

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

        // now should have two lists: head and fast

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

        ListNode p1 = head;
        ListNode p2 = second;

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

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

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

```
public static ListNode reverseOrder(ListNode head) {  
  
    if (head == null || head.next == null) {  
        return head;  
    }  
  
    ListNode pre = head;  
    ListNode curr = head.next;  
  
    while (curr != null) {  
        ListNode temp = curr.next;  
        curr.next = pre;  
        pre = curr;  
        curr = temp;  
    }  
  
    // set head node's next  
    head.next = null;  
  
    return pre;  
}  
  
public static void printList(ListNode n) {  
    System.out.println("-----");  
    while (n != null) {  
        System.out.print(n.val);  
        n = n.next;  
    }  
    System.out.println();  
}
```

---

### 110.3 Takeaway Messages

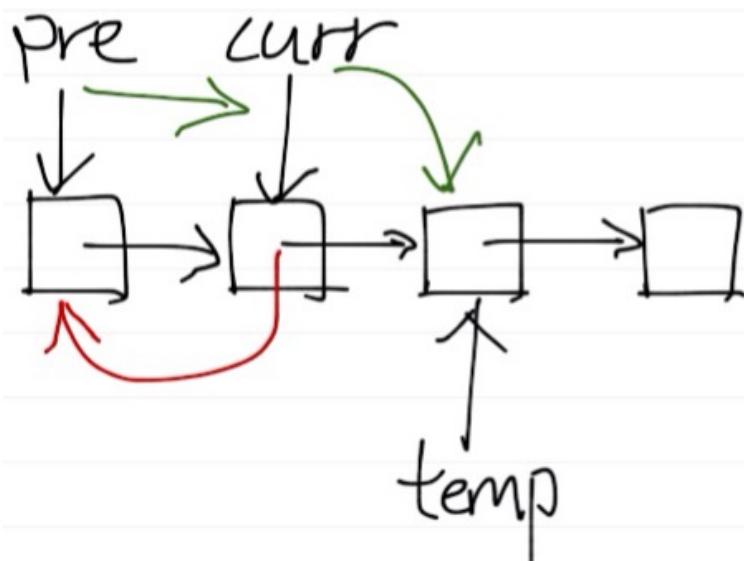
The three steps can be used to solve other problems of linked list. A little diagram may help better understand them.

Reverse List:

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

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

head.next = null;
```



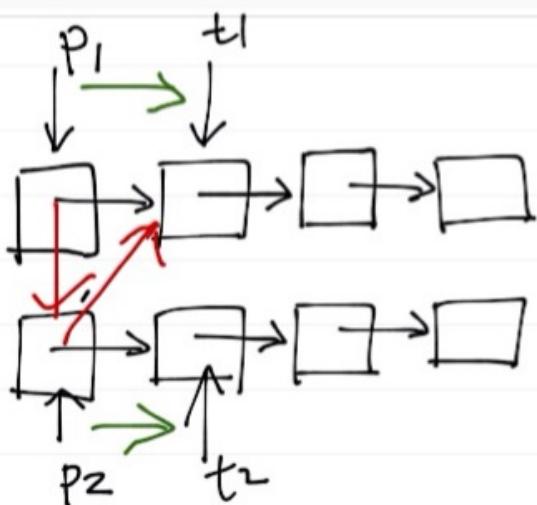
Merge List:

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

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

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

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



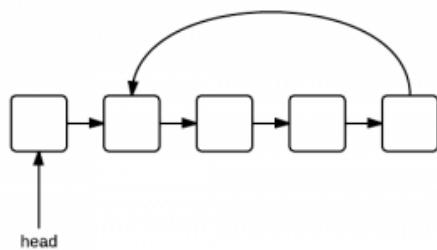


# 111 Linked List Cycle

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

## 111.1 Analysis

If we have 2 pointers - fast and slow. It is guaranteed that the fast one will meet the slow one if there exists a circle.



## 111.2 Java Solution

---

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

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

            if(slow == fast)
                return true;
        }

        return false;
    }
}
```

---



# 112 Copy List with Random Pointer

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.

## 112.1 Java Solution 1

We can solve this problem by doing the following steps:

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

---

```
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;
}
```

---

## 112.2 Java Solution 2 - Using HashMap

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

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

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

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

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

```
q.random = null;  
  
p = p.next;  
q = q.next;  
}  
  
return newHead;  
}
```

---



# 113 Merge Two Sorted Lists

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.

## 113.1 Analysis

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.

## 113.2 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;
            }
        }
    }
}
```

## 113 Merge Two Sorted Lists

---

```
    p = p.next;
}

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

return fakeHead.next;
}
}
```

---

## 114 Odd Even Linked List

### 114.1 Problem

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

The program should run in  $O(1)$  space complexity and  $O(\text{nodes})$  time complexity.

Example:

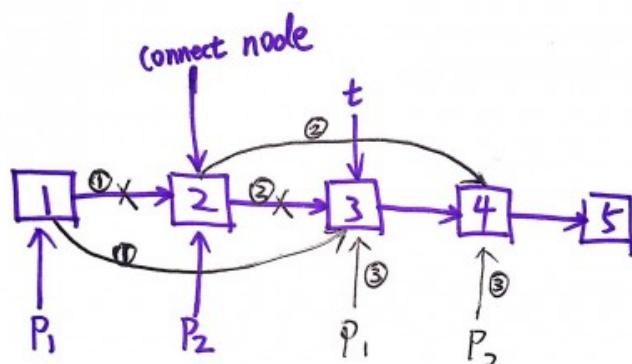
---

```
Given 1->2->3->4->5->NULL,  
return 1->3->5->2->4->NULL.
```

---

### 114.2 Analysis

This problem can be solved by using two pointers. We iterate over the link and move the two pointers.



### 114.3 Java Solution

---

```
public ListNode oddEvenList(ListNode head) {  
    if(head == null)  
        return head;  
  
    ListNode result = head;  
    ListNode p1 = head;  
    ListNode p2 = head.next;  
    ListNode connectNode = head.next;
```

```
while(p1 != null && p2 != null){  
    ListNode t = p2.next;  
    if(t == null)  
        break;  
  
    p1.next = p2.next;  
    p1 = p1.next;  
  
    p2.next = p1.next;  
    p2 = p2.next;  
}  
  
p1.next = connectNode;  
  
return result;  
}
```

---

# 115 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.

---

## 115.1 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.

## 115.2 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;
}
}
```

---

### 115.3 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;
    }
}
```

---

# 116 Remove Duplicates from Sorted List

## II

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

For example, given 1->1->1->2->3, return 2->3.

### 116.1 Java Solution

---

```
public ListNode deleteDuplicates(ListNode head) {
    ListNode t = new ListNode(0);
    t.next = head;

    ListNode p = t;
    while(p.next!=null&&p.next.next!=null){
        if(p.next.val == p.next.next.val){
            int dup = p.next.val;
            while(p.next!=null&&p.next.val==dup){
                p.next = p.next.next;
            }
        }else{
            p=p.next;
        }
    }
    return t.next;
}
```

---



# 117 Partition List

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

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

For example, given  $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow 2$  and  $x = 3$ , return  $1 \rightarrow 2 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5$ .

## 117.1 Java Solution

---

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

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

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

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

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

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

        // close the list
        p2.next = null;

        prev.next = fakeHead2.next;

        return fakeHead1.next;
```

```
    }  
}
```

---

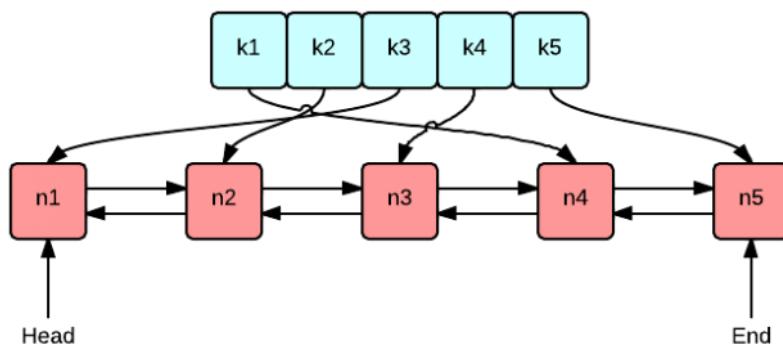
# 118 LRU Cache

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

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

## 118.1 Analysis

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



The LRU cache is a hash table of keys and double linked nodes. The hash table makes the time of get() to be O(1). The list of double linked nodes make the nodes adding/removal operations O(1).

## 118.2 Java Solution

Define a double linked list node.

```
class Node{
    int key;
    int value;
    Node pre;
    Node next;
```

## 118 LRU Cache

---

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


---



```
public class LRUCache {  
    int capacity;  
    HashMap<Integer, Node> map = new HashMap<Integer, Node>();  
    Node head=null;  
    Node end=null;  
  
    public LRUCache(int capacity) {  
        this.capacity = capacity;  
    }  
  
    public int get(int key) {  
        if(map.containsKey(key)){  
            Node n = map.get(key);  
            remove(n);  
            setHead(n);  
            return n.value;  
        }  
  
        return -1;  
    }  
  
    public void remove(Node n){  
        if(n.pre!=null){  
            n.pre.next = n.next;  
        }else{  
            head = n.next;  
        }  
  
        if(n.next!=null){  
            n.next.pre = n.pre;  
        }else{  
            end = n.pre;  
        }  
  
    }  
  
    public void setHead(Node n){  
        n.next = head;  
        n.pre = null;  
  
        if(head!=null)  
            head.pre = n;  
    }  
}
```


```

```
head = n;

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

public void set(int key, int value) {
    if(map.containsKey(key)){
        Node old = map.get(key);
        old.value = value;
        remove(old);
        setHead(old);
    }else{
        Node created = new Node(key, value);
        if(map.size()>=capacity){
            map.remove(end.key);
            remove(end);
            setHead(created);
        }else{
            setHead(created);
        }
        map.put(key, created);
    }
}
}
```

---



# 119 Intersection of Two Linked Lists

## 119.1 Problem

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

For example, the following two linked lists:

---

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

---

begin to intersect at node c1.

## 119.2 Java Solution

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

---

```
/*
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        int len1 = 0;
        int len2 = 0;
        ListNode p1=headA, p2=headB;
        if (p1 == null || p2 == null)
            return null;

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

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

        if (len1 > len2)
            p1 = p1.next;
        else
            p2 = p2.next;

        while(p1 != null && p2 != null){
            if (p1 == p2)
                return p1;
            p1 = p1.next;
            p2 = p2.next;
        }
    }
}
```

## 119 Intersection of Two Linked Lists

---

```
}

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

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

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

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

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

return null;
}
}
```

---

# 120 Remove Linked List Elements

Remove all elements from a linked list of integers that have value val.

Example

---

Given: 1 --> 2 --> 6 --> 3 --> 4 --> 5 --> 6, val = 6

Return: 1 --> 2 --> 3 --> 4 --> 5

---

## 120.1 Java Solution

The key to solve this problem is using a helper node to track the head of the list.

---

```
public ListNode removeElements(ListNode head, int val) {
    ListNode helper = new ListNode(0);
    helper.next = head;
    ListNode p = helper;

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

    return helper.next;
}
```

---



# 121 Swap Nodes in Pairs

Given a linked list, swap every two adjacent nodes and return its head.

For example, given `1->2->3->4`, you should return the list as `2->1->4->3`.

Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

## 121.1 Java Solution

Use two template variable to track the previous and next node of each pair.

---

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

    ListNode h = new ListNode(0);
    h.next = head;
    ListNode p = h;

    while(p.next != null && p.next.next != null){
        //use t1 to track first node
        ListNode t1 = p;
        p = p.next;
        t1.next = p.next;

        //use t2 to track next node of the pair
        ListNode t2 = p.next.next;
        p.next.next = p;
        p.next = t2;
    }

    return h.next;
}
```

---



## 122 Reverse Linked List

Reverse a singly linked list.

### 122.1 Java Solution 1 - Iterative

---

```
public ListNode reverseList(ListNode head) {  
    if(head==null || head.next == null)  
        return head;  
  
    ListNode p1 = head;  
    ListNode p2 = head.next;  
  
    head.next = null;  
    while(p1!= null && p2!= null){  
        ListNode t = p2.next;  
        p2.next = p1;  
        p1 = p2;  
        if (t!=null){  
            p2 = t;  
        }else{  
            break;  
        }  
    }  
  
    return p2;  
}
```

---

### 122.2 Java Solution 2 - Recursive

---

```
public ListNode reverseList(ListNode head) {  
    if(head==null || head.next == null)  
        return head;  
  
    //get second node  
    ListNode second = head.next;  
    //set first's next to be null  
    head.next = null;  
  
    ListNode rest = reverseList(second);
```

---

## 122 Reverse Linked List

---

```
second.next = head;  
  
return rest;  
}
```

---

# 123 Reverse Linked List II

Reverse a linked list from position m to n. Do it in-place and in one-pass.

For example: given 1->2->3->4->5->NULL, m = 2 and n = 4, return 1->4->3->2->5->NULL.

## 123.1 Analysis

## 123.2 Java Solution

---

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

    ListNode prev = null;//track (m-1)th node
    ListNode first = new ListNode(0); //first's next points to mth
    ListNode second = new ListNode(0); //second's next points to (n+1)th

    int i=0;
    ListNode p = head;
    while(p!=null){
        i++;
        if(i==m-1){
            prev = p;
        }

        if(i==m){
            first.next = p;
        }

        if(i==n){
            second.next = p.next;
            p.next = null;
        }

        p= p.next;
    }
    if(first.next == null)
        return head;

    // reverse list [m, n]
    ListNode p1 = first.next;
    ListNode p2 = p1.next;
```

## 123 Reverse Linked List II

---

```
p1.next = second.next;

while(p1!=null && p2!=null){
    ListNode t = p2.next;
    p2.next = p1;
    p1 = p2;
    p2 = t;
}

//connect to previous part
if(prev!=null)
    prev.next = p1;
else
    return p1;

return head;
}
```

---

## 124 Remove Nth Node From End of List

Given a linked list, remove the nth node from the end of list and return its head.

For example, given linked list 1->2->3->4->5 and n = 2, the result is 1->2->3->5.

### 124.1 Java Solution 1 - Naive Two Passes

Calculate the length first, and then remove the nth from the beginning.

---

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

    //get length of list
    ListNode p = head;
    int len = 0;
    while(p != null){
        len++;
        p = p.next;
    }

    //if remove first node
    int fromStart = len-n+1;
    if(fromStart==1)
        return head.next;

    //remove non-first node
    p = head;
    int i=0;
    while(p!=null){
        i++;
        if(i==fromStart-1){
            p.next = p.next.next;
        }
        p=p.next;
    }

    return head;
}
```

---

## 124.2 Java Solution 2 - One Pass

Use fast and slow pointers. The fast pointer is n steps ahead of the slow pointer. When the fast reaches the end, the slow pointer points at the previous element of the target element.

---

```
public ListNode removeNthFromEnd(ListNode head, int n) {  
    if(head == null)  
        return null;  
  
    ListNode fast = head;  
    ListNode slow = head;  
  
    for(int i=0; i<n; i++){  
        fast = fast.next;  
    }  
  
    //if remove the first node  
    if(fast == null){  
        head = head.next;  
        return head;  
    }  
  
    while(fast.next != null){  
        fast = fast.next;  
        slow = slow.next;  
    }  
  
    slow.next = slow.next.next;  
  
    return head;  
}
```

---

# 125 Implement Stack using Queues

Implement the following operations of a stack using queues. push(x) – Push element x onto stack. pop() – Removes the element on top of the stack. top() – Get the top element. empty() – Return whether the stack is empty.

Note: only standard queue operations are allowed, i.e., poll(), offer(), peek(), size() and isEmpty() in Java.

## 125.1 Analysis

This problem can be solved by using two queues.

## 125.2 Java Solution

---

```
class MyStack {
    LinkedList<Integer> queue1 = new LinkedList<Integer>();
    LinkedList<Integer> queue2 = new LinkedList<Integer>();

    // Push element x onto stack.
    public void push(int x) {
        if(empty()){
            queue1.offer(x);
        }else{
            if(queue1.size()>0){
                queue2.offer(x);
                int size = queue1.size();
                while(size>0){
                    queue2.offer(queue1.poll());
                    size--;
                }
            }else if(queue2.size()>0){
                queue1.offer(x);
                int size = queue2.size();
                while(size>0){
                    queue1.offer(queue2.poll());
                    size--;
                }
            }
        }
    }
}
```

## 125 Implement Stack using Queues

---

```
// Removes the element on top of the stack.  
public void pop() {  
    if(queue1.size()>0){  
        queue1.poll();  
    }else if(queue2.size()>0){  
        queue2.poll();  
    }  
}  
  
// Get the top element.  
public int top() {  
    if(queue1.size()>0){  
        return queue1.peek();  
    }else if(queue2.size()>0){  
        return queue2.peek();  
    }  
    return 0;  
}  
  
// Return whether the stack is empty.  
public boolean empty() {  
    return queue1.isEmpty() & queue2.isEmpty();  
}  
}
```

---

## 126 Implement Queue using Stacks

Implement the following operations of a queue using stacks.

push(x) – Push element x to the back of queue. pop() – Removes the element from in front of queue. peek() – Get the front element. empty() – Return whether the queue is empty.

### 126.1 Java Solution

---

```
class MyQueue {  
  
    Stack<Integer> temp = new Stack<Integer>();  
    Stack<Integer> value = new Stack<Integer>();  
  
    // Push element x to the back of queue.  
    public void push(int x) {  
        if(value.isEmpty()){  
            value.push(x);  
        }else{  
            while(!value.isEmpty()){  
                temp.push(value.pop());  
            }  
  
            value.push(x);  
  
            while(!temp.isEmpty()){  
                value.push(temp.pop());  
            }  
        }  
    }  
  
    // Removes the element from in front of queue.  
    public void pop() {  
        value.pop();  
    }  
  
    // Get the front element.  
    public int peek() {  
        return value.peek();  
    }  
  
    // Return whether the queue is empty.  
    public boolean empty() {  
        return value.isEmpty();  
    }  
}
```

## 126 Implement Queue using Stacks

---

```
public boolean empty() {  
    return value.isEmpty();  
}  
}
```

---

# 127 Palindrome Linked List

Given a singly linked list, determine if it is a palindrome.

## 127.1 Java Solution 1

We can create a new list in reversed order and then compare each node. The time and space are  $O(n)$ .

---

```
public boolean isPalindrome(ListNode head) {
    if(head == null)
        return true;

    ListNode p = head;
    ListNode prev = new ListNode(head.val);

    while(p.next != null){
        ListNode temp = new ListNode(p.next.val);
        temp.next = prev;
        prev = temp;
        p = p.next;
    }

    ListNode p1 = head;
    ListNode p2 = prev;

    while(p1!=null){
        if(p1.val != p2.val)
            return false;

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

    return true;
}
```

---

## 127.2 Java Solution 2

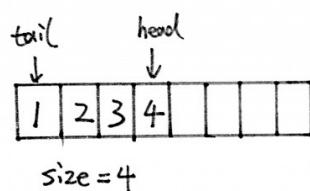
We can use a fast and slow pointer to get the center of the list, then reverse the second list and compare two sublists. The time is  $O(n)$  and space is  $O(1)$ .

```
public boolean isPalindrome(ListNode head) {  
  
    if(head == null || head.next==null)  
        return true;  
  
    //find list center  
    ListNode fast = head;  
    ListNode slow = head;  
  
    while(fast.next!=null && fast.next.next!=null){  
        fast = fast.next.next;  
        slow = slow.next;  
    }  
  
    ListNode secondHead = slow.next;  
    slow.next = null;  
  
    //reverse second part of the list  
    ListNode p1 = secondHead;  
    ListNode p2 = p1.next;  
  
    while(p1!=null && p2!=null){  
        ListNode temp = p2.next;  
        p2.next = p1;  
        p1 = p2;  
        p2 = temp;  
    }  
  
    secondHead.next = null;  
  
    //compare two sublists now  
    ListNode p = (p2==null?p1:p2);  
    ListNode q = head;  
    while(p!=null){  
        if(p.val != q.val)  
            return false;  
  
        p = p.next;  
        q = q.next;  
    }  
  
    return true;  
}
```

---

## 128 Implement a Queue using an Array in Java

The following Java code shows how to implement a queue without using any extra data structures in Java. We can implement a queue by using an array.



---

```
import java.lang.reflect.Array;
import java.util.Arrays;

public class Queue<E> {

    E[] arr;
    int head = -1;
    int tail = -1;
    int size;

    public Queue(Class<E> c, int size) {
        E[] newInstance = (E[]) Array.newInstance(c, size);
        this.arr = newInstance;
        this.size = 0;
    }

    boolean push(E e) {
        if (size == arr.length)
            return false;

        head = (head + 1) % arr.length;
        arr[head] = e;
        size++;

        if(tail == -1){
            tail = head;
        }
    }
}
```

## 128 Implement a Queue using an Array in Java

---

```
        return true;
    }

    boolean pop() {
        if (size == 0) {
            return false;
        }

        E result = arr[tail];
        arr[tail] = null;
        size--;
        tail = (tail+1)%arr.length;

        if (size == 0) {
            head = -1;
            tail = -1;
        }

        return true;
    }

    E peek(){
        if(size==0)
            return null;

        return arr[tail];
    }

    public int size() {
        return this.size;
    }

    public String toString() {
        return Arrays.toString(this.arr);
    }

    public static void main(String[] args) {
        Queue<Integer> q = new Queue<Integer>(Integer.class, 5);
        q.push(1);
        q.push(2);
        q.push(3);
        q.push(4);
        q.push(5);
        q.pop();
        q.push(6);
        System.out.println(q);
    }
}
```

---

## 129 Delete Node in a Linked List

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Supposed the linked list is `1 -> 2 -> 3 -> 4` and you are given the third node with value `3`, the linked list should become `1 -> 2 -> 4` after calling your function.

### 129.1 Java Solution

---

```
public void deleteNode(ListNode node) {  
    node.val = node.next.val;  
    node.next = node.next.next;  
}
```

---

Is this problem too easy? Or I'm doing it wrong.



# 130 Moving Average from Data Stream

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

## 130.1 Java Solution

This problem is solved by using a queue.

---

```
public class MovingAverage {

    LinkedList<Integer> queue;
    int size;

    /** Initialize your data structure here. */
    public MovingAverage(int size) {
        this.queue = new LinkedList<Integer>();
        this.size = size;
    }

    public double next(int val) {
        queue.offer(val);
        if(queue.size()>this.size){
            queue.poll();
        }
        int sum=0;
        for(int i: queue){
            sum=sum+i;
        }

        return (double)sum/queue.size();
    }
}
```

---



# 131 Java PriorityQueue Class Example

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

## 131.1 Simple Example

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

---

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

public class PriorityQueueTest {

    static class PQsort implements Comparator<Integer> {

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

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

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

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

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

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

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

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

}
}
```

---

Output:

---

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

---

## 131.2 Example of Solving Problems Using PriorityQueue

Merging k sorted list.

For more details about PriorityQueue, please go to [doc](#).

# 132 Binary Tree Preorder Traversal

## 132.1 Analysis

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

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

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

## 132.2 Java Solution

---

```
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;
}
}
```

---

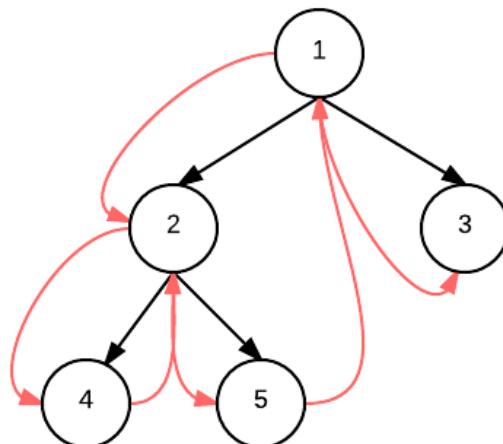
## 133 Binary Tree Inorder Traversal

There are 3 solutions for solving this problem.

### 133.1 Java Solution 1 - Iterative

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

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



---

```
//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.
    }
}
```

## 133 Binary Tree Inorder Traversal

---

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

if(root == null)
    return lst;

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

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

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

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

return lst;
}
```

---

## 133.2 Java Solution 2 - Recursive

The recursive solution is trivial.

---

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

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

        return result;
    }

    public void helper(TreeNode p){
        if(p.left!=null)
```

```
    helper(p.left);

    result.add(p.val);

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

---

### 133.3 Java Solution 3 - Simple

Updated on 4/28/2016

---

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

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

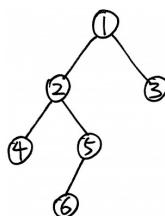
    return result;
}
```

---



## 134 Binary Tree Postorder Traversal

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



### 134.1 Java Solution 1

The key to iterative postorder traversal is the following:

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

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

---

```
//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;
}
```

---

## 134.2 Java Solution 2 - Simple!

Thanks to Edmond. This solution is superior!

---

```
public List<Integer> postorderTraversal(TreeNode root) {  
    List<Integer> res = new ArrayList<Integer>();  
  
    if(root==null) {  
        return res;  
    }  
  
    Stack<TreeNode> stack = new Stack<TreeNode>();  
    stack.push(root);  
  
    while(!stack.isEmpty()) {  
        TreeNode temp = stack.peek();  
        if(temp.left==null && temp.right==null) {  
            TreeNode pop = stack.pop();  
            res.add(pop.val);  
        }  
        else {  
            if(temp.right!=null) {  
                stack.push(temp.right);  
                temp.right = null;  
            }  
            if(temp.left!=null) {  
                stack.push(temp.left);  
                temp.left = null;  
            }  
        }  
    }  
  
    return res;  
}
```

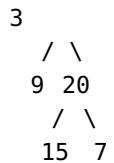
---



# 135 Binary Tree Level Order Traversal

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

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



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

## 135.1 Analysis

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

## 135.2 Java Solution

---

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

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

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

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

        nodeValues.add(node.val);
    }

    al.add(nodeValues);
    nodeValues.clear();
    current = next;
    next = new LinkedList<TreeNode>();
}
```

---

## 135 Binary Tree Level Order Traversal

---

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

return al;
}
```

---

## 136 Binary Tree Level Order Traversal II

Given a binary tree, return the bottom-up level order traversal of its nodes' values.

For example, given binary tree 3,9,20,#,#,15,7,

---

3

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

---

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

### 136.1 Java Solution

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

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

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

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

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

        numberList.add(head.val);

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

    if(current.isEmpty()){
        current = next;
    }
}
```

```
        next = new LinkedList<TreeNode>();
        result.add(numberList);
        numberList = new ArrayList<Integer>();
    }
}

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

return reversedResult;
}
```

---

# 137 Binary Tree Vertical Order Traversal

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

## 137.1 Java Solution

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

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

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

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

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

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

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

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

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

## 137 Binary Tree Vertical Order Traversal

---

```
        map.put(level, t);
    }

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

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

}

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

return result;
}
```

---

# 138 Invert Binary Tree

*Google: 90*

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

## 138.1 Java Solution 1 - Recursive

---

```
public TreeNode invertTree(TreeNode root) {  
    if(root!=null){  
        helper(root);  
    }  
  
    return root;  
}  
  
public void helper(TreeNode p){  
  
    TreeNode temp = p.left;  
    p.left = p.right;  
    p.right = temp;  
  
    if(p.left!=null)  
        helper(p.left);  
  
    if(p.right!=null)  
        helper(p.right);  
}
```

---

## 138.2 Java Solution 2 - Iterative

---

```
public TreeNode invertTree(TreeNode root) {  
    LinkedList<TreeNode> queue = new LinkedList<TreeNode>();  
  
    if(root!=null){  
        queue.add(root);  
    }  
  
    while(!queue.isEmpty()){  
        TreeNode p = queue.poll();  
        if(p.left!=null)
```

## 138 Invert Binary Tree

---

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

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

    return root;
}
```

---

# 139 Kth Smallest Element in a BST

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

## 139.1 Java Solution 1 - Inorder Traversal

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

---

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

    TreeNode p = root;
    int result = 0;

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

    return result;
}
```

---

## 139.2 Java Solution 2 - Extra Data Structure

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

coming soon...

---

---



# 140 Binary Tree Longest Consecutive Sequence

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

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

## 140.1 Java Solution 1 - Queue

```
public int longestConsecutive(TreeNode root) {  
    if(root==null)  
        return 0;  
  
    LinkedList<TreeNode> nodeQueue = new LinkedList<TreeNode>();  
    LinkedList<Integer> sizeQueue = new LinkedList<Integer>();  
  
    nodeQueue.offer(root);  
    sizeQueue.offer(1);  
    int max=1;  
  
    while(!nodeQueue.isEmpty()) {  
        TreeNode head = nodeQueue.poll();  
        int size = sizeQueue.poll();  
  
        if(head.left!=null){  
            int leftSize=size;  
            if(head.val==head.left.val-1){  
                leftSize++;  
                max = Math.max(max, leftSize);  
            }else{  
                leftSize=1;  
            }  
  
            nodeQueue.offer(head.left);  
            sizeQueue.offer(leftSize);  
        }  
  
        if(head.right!=null){  
            int rightSize=size;  
            if(head.val==head.right.val-1){  
                rightSize++;  
                max = Math.max(max, rightSize);  
            }else{  
                rightSize=1;  
            }  
  
            nodeQueue.offer(head.right);  
            sizeQueue.offer(rightSize);  
        }  
    }  
}
```

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

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

}

return max;
}
```

---

## 140.2 Java Solution 2 - Recursion

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

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

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

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

        int fromLeft = 0;
        int fromRight= 0;

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

        if(root.right==null){
            fromRight=1;
        }else if(root.right.val-1==root.val){
```

```
    fromRight = r+1;
}else{
    fromRight=1;
}

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

return Math.max(fromLeft, fromRight);
}

}
```

---



# 141 Validate Binary Search Tree

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

Assume a BST is defined as follows:

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

## 141.1 Java Solution 1 - Recursive

All values on the left sub tree must be less than root, and all values on the right sub tree must be greater than root. So we just check the boundaries for each node.

---

```
public boolean isValidBST(TreeNode root) {  
    return isValidBST(root, Double.NEGATIVE_INFINITY,  
                      Double.POSITIVE_INFINITY);  
}  
  
public boolean isValidBST(TreeNode p, double min, double max){  
    if(p==null)  
        return true;  
  
    if(p.val <= min || p.val >= max)  
        return false;  
  
    return isValidBST(p.left, min, p.val) && isValidBST(p.right, p.val, max);  
}
```

---

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

## 141.2 Java Solution 2 - Iterative

---

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

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

---

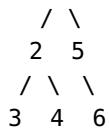
## 142 Flatten Binary Tree to Linked List

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

For example, Given

---

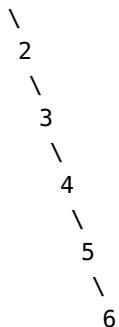
1



The flattened tree should look like:

---

1



### 142.1 Thoughts

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

### 142.2 Java Solution

---

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

## 142 Flatten Binary Tree to Linked List

---

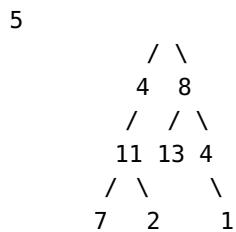
```
public void flatten(TreeNode root) {  
    Stack<TreeNode> stack = new Stack<TreeNode>();  
    TreeNode p = root;  
  
    while(p != null || !stack.empty()){  
  
        if(p.right != null){  
            stack.push(p.right);  
        }  
  
        if(p.left != null){  
            p.right = p.left;  
            p.left = null;  
        }else if(!stack.empty()){  
            TreeNode temp = stack.pop();  
            p.right=temp;  
        }  
  
        p = p.right;  
    }  
}
```

---

## 143 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,



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

### 143.1 Java Solution 1 - Using Queue

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

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

---

```
/*
 * 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 currNode = nodes.remove();
            Integer currValue = values.remove();

            if(currNode.left == null && currNode.right == null && currValue == sum)
                return true;

            if(currNode.left != null)
                nodes.add(currNode.left);
                values.add(currValue + currNode.left.val);

            if(currNode.right != null)
                nodes.add(currNode.right);
                values.add(currValue + currNode.right.val);
        }

        return false;
    }
}
```

## 143 Path Sum

---

```
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;
}
}
```

---

## 143.2 Java Solution 2 - Recursion

---

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

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

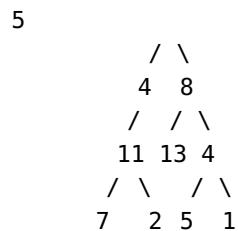
---

Thanks to nebulaliang, this solution is wonderful!

## 144 Path Sum II

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

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



the method returns the following:

---

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

---

### 144.1 Analysis

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

### 144.2 Java Solution

---

```
public List<ArrayList<Integer>> pathSum(TreeNode root, int sum) {  
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();  
    if(root == null)  
        return result;  
  
    ArrayList<Integer> l = new ArrayList<Integer>();  
    l.add(root.val);  
    dfs(root, sum-root.val, result, l);  
    return result;  
}
```

---

## 144 Path Sum II

---

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

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

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

---

# 145 Construct Binary Tree from Inorder and Postorder Traversal

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

## 145.1 Analysis

This problem can be illustrated by using a simple example.

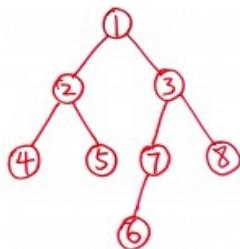
---

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

---

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

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



## 145.2 Java Solution

---

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

---

## 145 Construct Binary Tree from Inorder and Postorder Traversal

---

```
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));
// Because k is not the length, 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;
}
```

---

## 146 Construct Binary Tree from Preorder and Inorder Traversal

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

### 146.1 Analysis

Consider the following example:

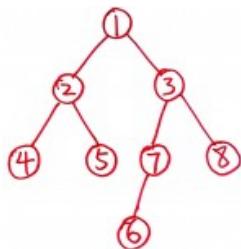
---

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

---

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

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



### 146.2 Java Solution

---

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

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

public TreeNode construct(int[] preorder, int preStart, int preEnd, int[]
inorder, int inStart, int inEnd){
```

```
if(preStart>preEnd||inStart>inEnd){  
    return null;  
}  
  
int val = preorder[preStart];  
TreeNode p = new TreeNode(val);  
  
//find parent element index from inorder  
int k=0;  
for(int i=0; i<inorder.length; i++){  
    if(val == inorder[i]){  
        k=i;  
        break;  
    }  
}  
  
p.left = construct(preorder, preStart+1, preStart+(k-inStart), inorder,  
    inStart, k-1);  
p.right= construct(preorder, preStart+(k-inStart)+1, preEnd, inorder, k+1  
    , inEnd);  
  
return p;  
}
```

---

# 147 Convert Sorted Array to Binary Search Tree

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

## 147.1 Thoughts

Straightforward! Recursively do the job.

## 147.2 Java Solution

---

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

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

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

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

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

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

        return root;
    }
}
```

## 147 Convert Sorted Array to Binary Search Tree

---

```
    }  
}
```

---

# 148 Convert Sorted List to Binary Search Tree

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

## 148.1 Thoughts

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

## 148.2 Java Solution

---

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

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

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

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

public class Solution {
    static ListNode h;
```

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

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

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

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

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

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

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

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

    return root;
}
}
```

---

# 149 Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

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

## 149.1 Thoughts

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

## 149.2 Java Solution

---

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

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

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

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

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

## 149 Minimum Depth of Binary Tree

---

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

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

return 0;
}
```

---

# 150 Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree. For example, given the below binary tree

---

1



the result is 6.

## 150.1 Analysis

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

## 150.2 Java Solution

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

---

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

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

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

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

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

    return current;
}
```

---



# 151 Balanced Binary Tree

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

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

## 151.1 Analysis

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

## 151.2 Java Solution

---

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

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

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

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

        return true;
    }

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

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

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

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

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

## 151 Balanced Binary Tree

---

```
    return -1;

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

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

---

# 152 Symmetric Tree

## 152.1 Problem

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

For example, this binary tree is symmetric:

---

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

---

But the following is not:

---

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

---

## 152.2 Java Solution - Recursion

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

---

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

public boolean isSymmetric(TreeNode l, TreeNode r) {
    if (l == null && r == null)
        return true;
    } else if (r == null || l == null) {
        return false;
    }

    if (l.val != r.val)
```

## 152 Symmetric Tree

---

```
    return false;

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

    return true;
}
```

---

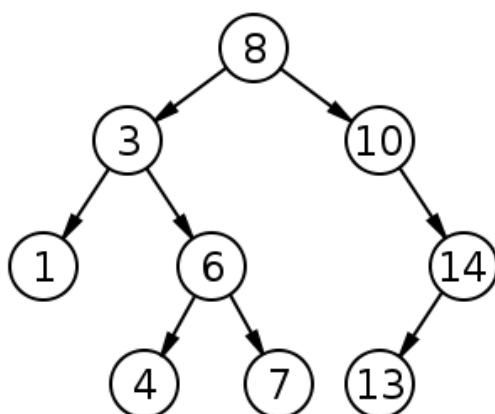
# 153 Binary Search Tree Iterator

## 153.1 Problem

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

## 153.2 Java Solution

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



---

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

public class BSTIterator {
    Stack<TreeNode> stack;
    public BSTIterator(TreeNode root) {
        // Initialize stack with root
        stack.push(root);
        // Move to leftmost node
        while (stack.peek().left != null) {
            stack.push(stack.peek().left);
        }
    }

    public int next() {
        // Pop current node from stack
        TreeNode current = stack.pop();
        // Move to its right child
        if (current.right != null) {
            stack.push(current.right);
            // Move to leftmost node in right subtree
            while (stack.peek().left != null) {
                stack.push(stack.peek().left);
            }
        }
        return current.val;
    }

    public boolean hasNext() {
        // Check if stack is empty
        return !stack.isEmpty();
    }
}
```

## 153 Binary Search Tree Iterator

---

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

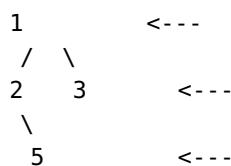
public boolean hasNext() {
    return !stack.isEmpty();
}

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

---

## 154 Binary Tree Right Side View

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



You can see [1, 3, 5].

### 154.1 Analysis

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

### 154.2 Java Solution

---

```
public List<Integer> rightSideView(TreeNode root) {  
    ArrayList<Integer> result = new ArrayList<Integer>();  
  
    if(root == null) return result;  
  
    LinkedList<TreeNode> queue = new LinkedList<TreeNode>();  
    queue.add(root);  
  
    while(queue.size() > 0){  
        //get size here  
        int size = queue.size();  
  
        for(int i=0; i<size; i++){  
            TreeNode top = queue.remove();  
  
            //the first element in the queue (right-most of the tree)  
            if(i==0){  
                result.add(top.val);  
            }  
            //add right first  
            if(top.right != null){
```

## 154 Binary Tree Right Side View

---

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

return result;
}
```

---

# 155 Lowest Common Ancestor of a Binary Search Tree

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

## 155.1 Analysis

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

## 155.2 Java Solution

---

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

---

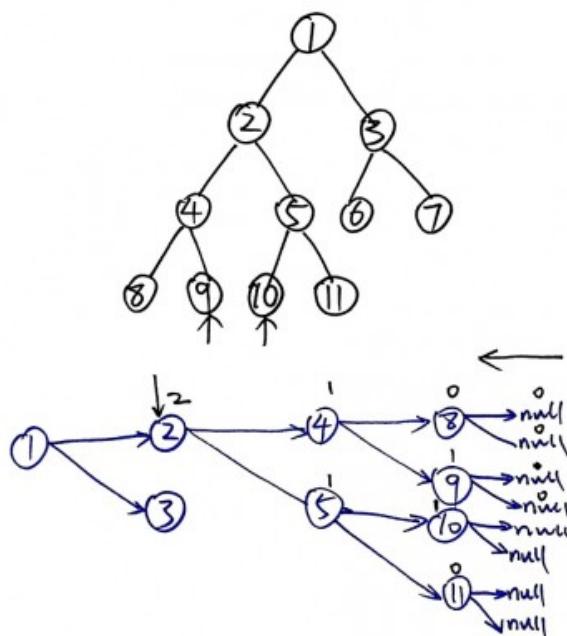


## 156 Lowest Common Ancestor of a Binary Tree

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

### 156.1 Java Solution 1

Please use the following diagram to walk through the solution.



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

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

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

```
}

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

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

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

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

        int numTotal = left.count + right.count;
        if(root== p || root == q){
            numTotal++;
        }
        return new Entity(numTotal, root);
    }
}
```

---

## 156.2 Java Solution 2

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

---

## 157 Verify Preorder Serialization of a Binary Tree

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

---

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

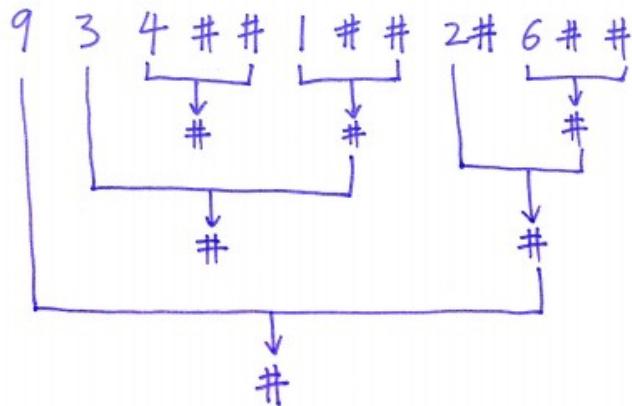
---

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

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

### 157.1 Java Solution - Stack

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



## 157 Verify Preorder Serialization of a Binary Tree

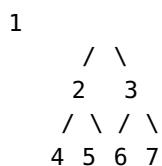
---

```
public boolean isValidSerialization(String preorder) {  
    LinkedList<String> stack = new LinkedList<String>();  
    String[] arr = preorder.split(",");  
  
    for(int i=0; i<arr.length; i++){  
        stack.add(arr[i]);  
  
        while(stack.size()>=3  
              && stack.get(stack.size()-1).equals("#")  
              && stack.get(stack.size()-2).equals("#")  
              && !stack.get(stack.size()-3).equals("#")){  
  
            stack.remove(stack.size()-1);  
            stack.remove(stack.size()-1);  
            stack.remove(stack.size()-1);  
  
            stack.add("#");  
        }  
  
        if(stack.size()==1 && stack.get(0).equals("#"))  
            return true;  
        else  
            return false;  
    }  
}
```

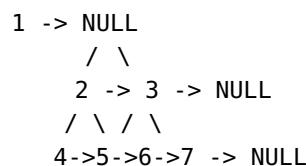
---

## 158 Populating Next Right Pointers in Each Node

Given the following perfect binary tree,

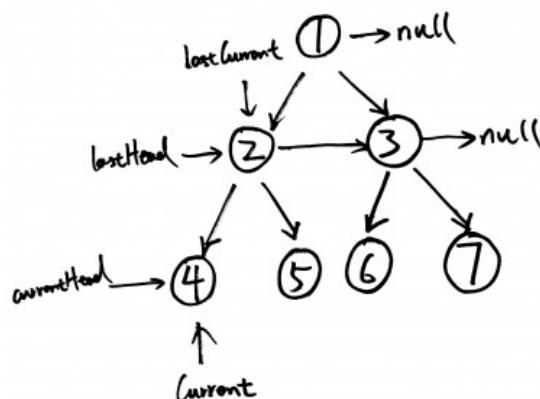


After calling your function, the tree should look like:



### 158.1 Java Solution

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



---

```
public void connect(TreeLinkNode root) {
```

## 158 Populating Next Right Pointers in Each Node

---

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

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

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

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

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

        lastCurrent = lastCurrent.next;
    }

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

}
```

---

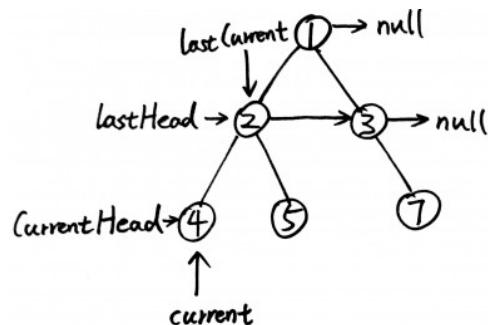
## 159 Populating Next Right Pointers in Each Node II

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

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

### 159.1 Analysis

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



### 159.2 Java Solution

```
public void connect(TreeLinkNode root) {  
    if(root == null)  
        return;  
  
    TreeLinkNode lastHead = root;//previous level's head  
    TreeLinkNode lastCurrent = null;//previous level's pointer  
    TreeLinkNode currentHead = null;//currnet level's head  
    TreeLinkNode current = null;//current level's pointer  
  
    while(lastHead!=null){  
        lastCurrent = lastHead;  
  
        while(lastCurrent!=null){  
            //left child is not null  
            if(lastCurrent.left!=null) {  
                lastCurrent.left.next = lastCurrent.right;  
                if(lastCurrent.right != null)  
                    lastCurrent.right.next = lastCurrent.next.left;  
            }  
            lastCurrent = lastCurrent.next;  
        }  
        lastHead = currentHead = lastCurrent;  
    }  
}
```

## 159 Populating Next Right Pointers in Each Node II

---

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

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

    lastCurrent = lastCurrent.next;
}

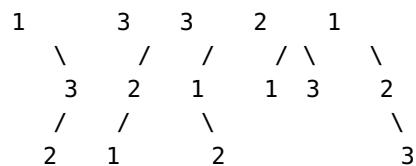
//update last head
lastHead = currentHead;
currentHead = null;
}
}
```

---

# 160 Unique Binary Search Trees

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

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



## 160.1 Analysis

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

---

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

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

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

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

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

...
...
...

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

---

Use dynamic programming to solve the problem.

## 160.2 Java Solution

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

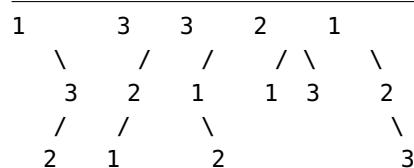
---

Check out how to get all unique binary search trees.

# 161 Unique Binary Search Trees II

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

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



## 161.1 Analysis

Check out Unique Binary Search Trees I.

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

## 161.2 Java Solution

---

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

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

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

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

---

## 161 Unique Binary Search Trees II

---

```
        node.left = left;
        node.right = right;
        list.add(node);
    }
}
}

return list;
}
```

---

## 162 Sum Root to Leaf Numbers

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

For example,



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

### 162.1 Java Solution - Recursive

This problem can be solved by a typical DFS approach.

---

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

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

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

    return result;
}

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

```
t.addAll(l);
all.add(t);
}

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

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

}
```

---

Same approach, but simpler coding style.

---

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

    return dfs(root, 0, 0);
}

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

    num = num*10 + node.val;

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

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

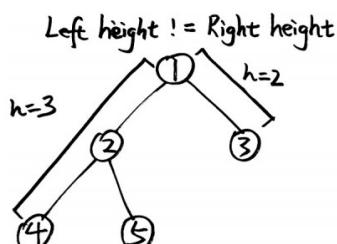
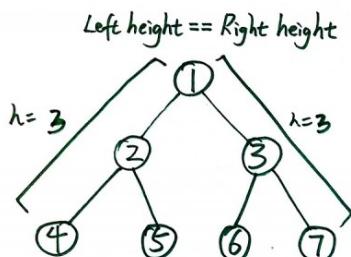
---

# 163 Count Complete Tree Nodes

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

## 163.1 Analysis

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



Time complexity is  $O(h^2)$ .

## 163.2 Java Solution

---

```
public int countNodes(TreeNode root) {  
    if(root==null)  
        return 0;  
  
    int left = getLeftHeight(root)+1;  
    int right = getRightHeight(root)+1;
```

## 163 Count Complete Tree Nodes

---

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

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

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

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

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

---

# 164 Closest Binary Search Tree Value

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

## 164.1 Java Solution

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

---

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

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

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

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

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

---



# 165 Binary Tree Paths

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

## 165.1 Java Solution

A typical depth-first search problem.

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

    if(root==null)
        return finalResult;

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

    dfs(root, results, curr);

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

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

    return finalResult;
}

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

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

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

## 165 Binary Tree Paths

---

```
    }

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

---

# 166 Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

## 166.1 Java Solution

---

```
public int maxDepth(TreeNode root) {  
    if(root==null)  
        return 0;  
  
    int leftDepth = maxDepth(root.left);  
    int rightDepth = maxDepth(root.right);  
  
    int bigger = Math.max(leftDepth, rightDepth);  
  
    return bigger+1;  
}
```

---



# 167 Recover Binary Search Tree

Two elements of a binary search tree (BST) are swapped by mistake. Recover the tree without changing its structure.

## 167.1 Java Solution

Inorder traversal will return values in an increasing order. So if an element is less than its previous element, the previous element is a swapped node.

```
public class Solution {
    TreeNode first;
    TreeNode second;
    TreeNode pre;

    public void inorder(TreeNode root){
        if(root==null)
            return;

        inorder(root.left);

        if(pre==null){
            pre=root;
        }else{
            if(root.val<pre.val){
                if(first==null){
                    first=pre;
                }

                second=root;
            }
            pre=root;
        }

        inorder(root.right);
    }

    public void recoverTree(TreeNode root) {
        if(root==null)
            return;

        inorder(root);
        if(second!=null && first !=null){

```

## 167 Recover Binary Search Tree

---

```
        int val = second.val;
        second.val = first.val;
        first.val = val;
    }

}
```

---

## 168 Merge K Sorted Arrays in Java

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

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

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

---

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

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

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

---

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

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

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

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

## 168 Merge K Sorted Arrays in Java

---

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

    return result;
}

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

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

---

# 169 Merge k Sorted Lists

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

## 169.1 Analysis

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

## 169.2 Java Solution

---

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

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

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

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

        //PriorityQueue is a sorted queue
        PriorityQueue<ListNode> q = new PriorityQueue<ListNode>(lists.size(),
            new Comparator<ListNode>() {
                public int compare(ListNode a, ListNode b) {
                    if (a.val > b.val)
                        return 1;
                    else if(a.val == b.val)
                        return 0;
                    else
                        return -1;
                }
            });
        for (int i = 0; i < lists.size(); i++) {
            if (lists.get(i) != null)
                q.add(lists.get(i));
        }

        ListNode head = null;
        ListNode tail = null;
        while (!q.isEmpty()) {
            ListNode node = q.poll();
            if (head == null)
                head = node;
            else
                tail.next = node;
            tail = node;
            if (node.next != null)
                q.add(node.next);
        }
        return head;
    }
}
```

## 169 Merge k Sorted Lists

---

```
        else
            return -1;
    }
});

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

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

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

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

    p = p.next;
}

return head.next;
}
}
```

---

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

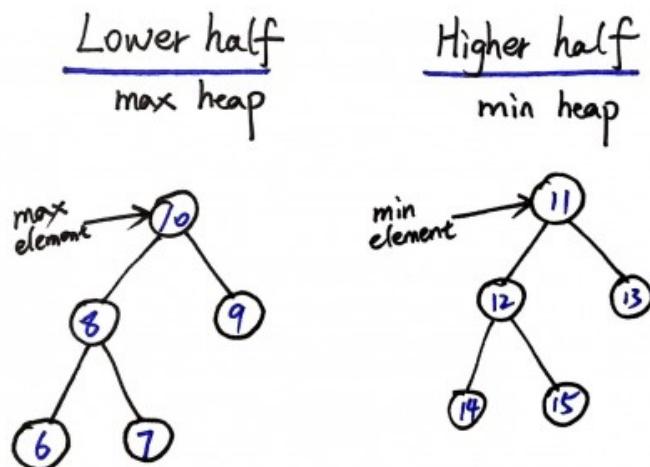
# 170 Find Median from Data Stream

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

## 170.1 Analysis

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

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



## 170.2 Java Solution

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

    public MedianFinder(){
        maxHeap = new PriorityQueue<Integer>(Collections.reverseOrder());
        minHeap = new PriorityQueue<Integer>();
    }
}
```

## 170 Find Median from Data Stream

---

```
// Adds a number into the data structure.
public void addNum(int num) {
    maxHeap.offer(num);
    minHeap.offer(maxHeap.poll());

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

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

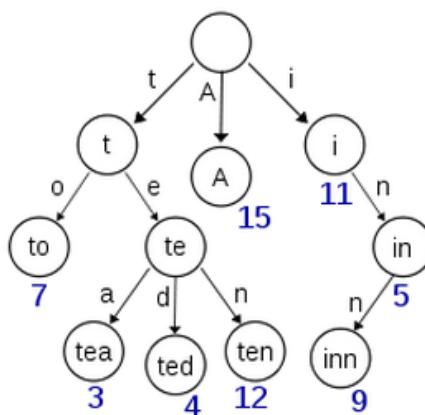
---

# 171 Implement Trie (Prefix Tree)

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

## 171.1 Java Solution 1

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



---

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

    public TrieNode() {}

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

public class Trie {
    private TrieNode root;

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

---

## 171 Implement Trie (Prefix Tree)

---

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

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

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

        children = t.children;

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

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

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

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

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

---

```

        children = t.children;
    }else{
        return null;
    }
}

return t;
}
}

```

---

## 171.2 Java Solution 2 - Improve Performance by Using an Array

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

---

```

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

public class Trie {
    private TrieNode root;

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

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

```

## 171 Implement Trie (Prefix Tree)

---

```
}

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

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

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

    if(p==root)
        return null;

    return p;
}
}
```

---

# 172 Add and Search Word Data structure design

Design a data structure that supports the following two operations:

---

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

---

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

## 172.1 Java Solution 1

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

TrieNode

---

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

    public TrieNode() {}

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

---

WordDictionary

---

```
public class WordDictionary {
    private TrieNode root;

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

    // Adds a word into the data structure.
    public void addWord(String word) {
        HashMap<Character, TrieNode> children = root.children;
```

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

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

    children = t.children;

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

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

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

    char c = word.charAt(start);

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

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

```
//if any path is true, set result to be true;
if(dfsSearch(child.getValue().children, word, start+1)){
    result = true;
}
}

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

---

## 172.2 Java Solution 2 - Using Array Instead of HashMap

---

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

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

public class WordDictionary {
    TrieNode root;

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

```
}

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

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

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

    char c = word.charAt(start);

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

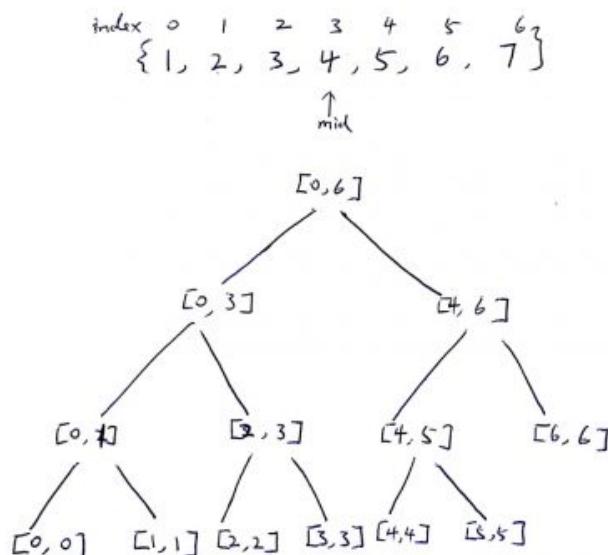
        if (p.arr[index] != null) {
            return dfsSearch(p.arr[index], word, start + 1);
        } else {
            return false;
        }
    }
    return false;
}
```

---

# 173 Range Sum Query Mutable

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

## 173.1 Java Solution



---

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

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

## 173 Range Sum Query Mutable

---

```
        this.end=right;
        this.sum=0;
    }
}

public class NumArray {
    TreeNode root = null;

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

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

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

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

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

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

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

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

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

        if(i<=root.start && j>=root.end){
            return root.sum;
        }
    }
}
```

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

    return result;
}

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

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

    TreeNode current = new TreeNode(i, j);

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

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

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

    return current;
}
}
```

---



# 174 The Skyline Problem

## 174.1 Analysis

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

## 174.2 Java Solution

---

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

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

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

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

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

    // add all left/right edges
    for (int[] building : buildings) {
        Edge startEdge = new Edge(building[0], building[2], true);
        edges.add(startEdge);
        Edge endEdge = new Edge(building[1], building[2], false);
        edges.add(endEdge);
    }

    // sort edges
```

---

```
Collections.sort(edges, new Comparator<Edge>() {
    public int compare(Edge a, Edge b) {
        if (a.x != b.x)
            return Integer.compare(a.x, b.x);

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

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

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

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

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

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

return result;
}
```

---

# 175 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.

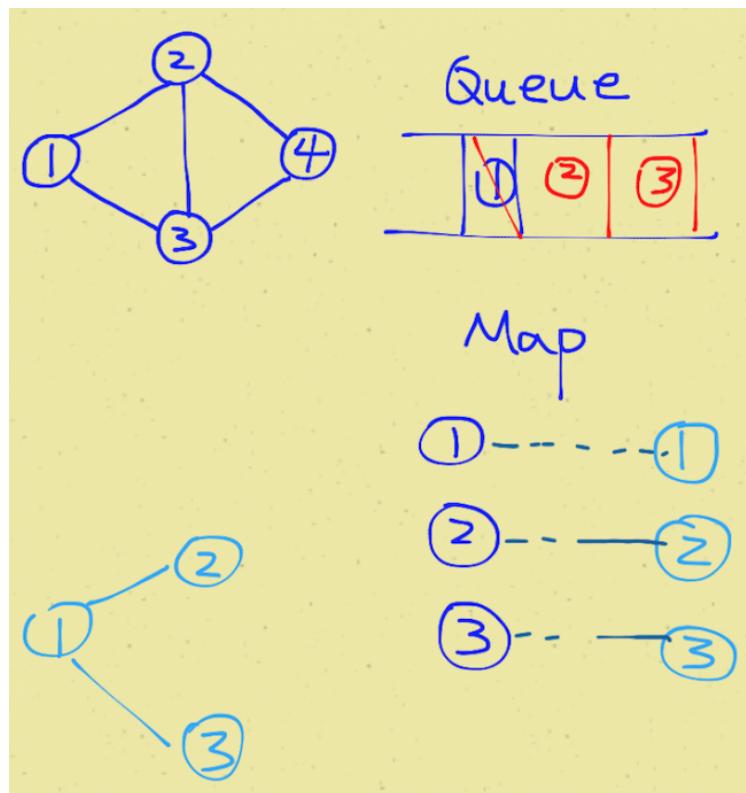
Visually, the graph looks like the following:



## 175.1 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;
}
```

---



# 176 Course Schedule

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

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

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

## 176.1 Analysis

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

## 176.2 Java Solution 1 - BFS

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

---

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

    int len = prerequisites.length;

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

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

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

    while(!queue.isEmpty()){
        int course = queue.remove();
        for(int i=0; i<len; i++){
            if(prerequisites[i][1] == course){
                pCounter[prerequisites[i][0]]--;
                if(pCounter[prerequisites[i][0]]==0){
                    queue.add(prerequisites[i][0]);
                }
            }
        }
    }

    return pCounter[0] == numCourses;
}
```

```
        }
    }

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

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

    return numNoPre == numCourses;
}
```

---

### 176.3 Java Solution 2 - DFS

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

    int len = prerequisites.length;

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

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

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

    for(int i=0; i<visit.length; i++){
        if(visit[i] == 0){
            if(dfs(i, visit, map)){
                return false;
            }
        }
    }

    return true;
}

private boolean dfs(int index, int[] visit, HashMap<Integer,ArrayList<Integer>> map){
    if(visit[index] == 1){
        return true;
    }

    if(visit[index] == 2){
        return false;
    }

    visit[index] = 1;

    for(Integer i: map.get(index)){
        if(dfs(i, visit, map)){
            return true;
        }
    }

    visit[index] = 2;
    return false;
}
```

```
    l.add(a[0]);
    map.put(a[1], l);
}
}

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

return true;
}

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

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

visit[i]=1;

return true;
}
```

---

Topological Sort Video from Coursera.



# 177 Course Schedule II

This is an extension of [Course Schedule](#). This time a valid sequence of courses is required as output.

## 177.1 Analysis

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

## 177.2 Java Solution

---

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

    int len = prerequisites.length;

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

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

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

## 177 Course Schedule II

---

```
int numNoPre = queue.size();

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

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

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

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

---

# 178 Reconstruct Itinerary

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

## 178.1 Analysis

This is an application of Hierholzer's algorithm to find a [Eulerian path](#).

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

## 178.2 Java Solution

---

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

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

        dfs("JFK");
        return result;
    }

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

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

        result.addFirst(s);
    }
}
```

---



# 179 Graph Valid Tree

Given  $n$  nodes labeled from  $0$  to  $n - 1$  and a list of undirected edges (each edge is a pair of nodes), check if these edges form a valid tree.

## 179.1 Analysis

This problem can be converted to finding a cycle in a graph. It can be solved by using DFS (Recursion) or BFS (Queue).

## 179.2 Java Solution 1 - DFS

---

```
public boolean validTree(int n, int[][] edges) {
    HashMap<Integer, ArrayList<Integer>> map = new HashMap<Integer,
        ArrayList<Integer>>();
    for(int i=0; i<n; i++){
        ArrayList<Integer> list = new ArrayList<Integer>();
        map.put(i, list);
    }

    for(int[] edge: edges){
        map.get(edge[0]).add(edge[1]);
        map.get(edge[1]).add(edge[0]);
    }

    boolean[] visited = new boolean[n];

    if(!helper(0, -1, map, visited))
        return false;

    for(boolean b: visited){
        if(!b)
            return false;
    }

    return true;
}

public boolean helper(int curr, int parent, HashMap<Integer,
    ArrayList<Integer>> map, boolean[] visited){
    if(visited[curr])
        return false;
```

```
visited[curr] = true;

for(int i: map.get(curr)){
    if(i!=parent && !helper(i, curr, map, visited)){
        return false;
    }
}

return true;
}
```

---

### 179.3 Java Solution 2 - BFS

```
public boolean validTree(int n, int[][][] edges) {
    HashMap<Integer, ArrayList<Integer>> map = new HashMap<Integer,
        ArrayList<Integer>>();
    for(int i=0; i<n; i++){
        ArrayList<Integer> list = new ArrayList<Integer>();
        map.put(i, list);
    }

    for(int[] edge: edges){
        map.get(edge[0]).add(edge[1]);
        map.get(edge[1]).add(edge[0]);
    }

    boolean[] visited = new boolean[n];

    LinkedList<Integer> queue = new LinkedList<Integer>();
    queue.offer(0);
    while(!queue.isEmpty()){
        int top = queue.poll();
        if(visited[top])
            return false;

        visited[top]=true;

        for(int i: map.get(top)){
            if(!visited[i])
                queue.offer(i);
        }
    }

    for(boolean b: visited){
        if(!b)
            return false;
    }
}
```

```
    }  
  
    return true;  
}
```

---



# 180 How Developers Sort in Java?

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

## 180.1 Using sort() Method

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

---

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

---

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

---

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

---

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

## 180.2 Using Sorted Data Structures

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

---

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

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

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

---

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

---

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

### 180.3 Bad Practices

There are still bad practices, such as using self-defined sorting algorithm. Take the code below for example, not only the algorithm is not efficient, but also it is not readable. This happens a lot in different forms of variations.

```
double t;
for (int i = 0; i < 2; i++)
    for (int j = i + 1; j < 3; j++)
        if (r[j] < r[i]) {
            t = r[i];
            r[i] = r[j];
            r[j] = t;
        }
```

---

# 181 Solution Merge Sort LinkedList in Java

LeetCode - Sort List:

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

## 181.1 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;
        }
    }
}
```

## 181 Solution Merge Sort LinkedList in Java

---

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

## 181 Solution Merge Sort LinkedList in Java

---

}

---

**Output:**

2 3 3 4 4 5

## 182 Quicksort Array in Java

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

The basic step of sorting an array are as follows:

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

[Here](#) is a very good animation of quicksort.

---

```
public class QuickSort {  
    public static void main(String[] args) {  
        int[] x = { 9, 2, 4, 7, 3, 7, 10 };  
        System.out.println(Arrays.toString(x));  
  
        int low = 0;  
        int high = x.length - 1;  
  
        quickSort(x, low, high);  
        System.out.println(Arrays.toString(x));  
    }  
  
    public static void quickSort(int[] arr, int low, int high) {  
        if (arr == null || arr.length == 0)  
            return;  
  
        if (low >= high)  
            return;  
  
        // pick the pivot  
        int middle = low + (high - low) / 2;  
        int pivot = arr[middle];  
  
        // make left < pivot and right > pivot  
        int i = low, j = high;  
        while (i <= j) {  
            while (arr[i] < pivot) {  
                i++;  
            }  
        }  
    }  
}
```

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

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

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

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

---

Output:

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

## 183 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



# 184 Maximum Gap

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Try to solve it in linear time/space. Return 0 if the array contains less than 2 elements. You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

## 184.1 Analysis

We can use a bucket-sort like algorithm to solve this problem in time of  $O(n)$  and space  $O(n)$ . The basic idea is to project each element of the array to an array of buckets. Each bucket tracks the maximum and minimum elements. Finally, scanning the bucket list, we can get the maximum gap.

The key part is to get the interval:

---

```
From: interval * (num[i] - min) = 0 and interval * (max - num[i]) = n
interval = num.length / (max - min)
```

---

See the internal comment for more details.

## 184.2 Java Solution

---

```
class Bucket{
    int low;
    int high;
    public Bucket(){
        low = -1;
        high = -1;
    }
}

public int maximumGap(int[] num) {
    if(num == null || num.length < 2){
        return 0;
    }

    int max = num[0];
    int min = num[0];
    for(int i=1; i<num.length; i++){
        max = Math.max(max, num[i]);
    }
```

```
    min = Math.min(min, num[i]);
}

// initialize an array of buckets
Bucket[] buckets = new Bucket[num.length+1]; //project to (0 - n)
for(int i=0; i<buckets.length; i++){
    buckets[i] = new Bucket();
}

double interval = (double) num.length / (max - min);
//distribute every number to a bucket array
for(int i=0; i<num.length; i++){
    int index = (int) ((num[i] - min) * interval);

    if(buckets[index].low == -1){
        buckets[index].low = num[i];
        buckets[index].high = num[i];
    }else{
        buckets[index].low = Math.min(buckets[index].low, num[i]);
        buckets[index].high = Math.max(buckets[index].high, num[i]);
    }
}

//scan buckets to find maximum gap
int result = 0;
int prev = buckets[0].high;
for(int i=1; i<buckets.length; i++){
    if(buckets[i].low != -1){
        result = Math.max(result, buckets[i].low-prev);
        prev = buckets[i].high;
    }
}

return result;
}
```

---

# 185 First Missing Positive

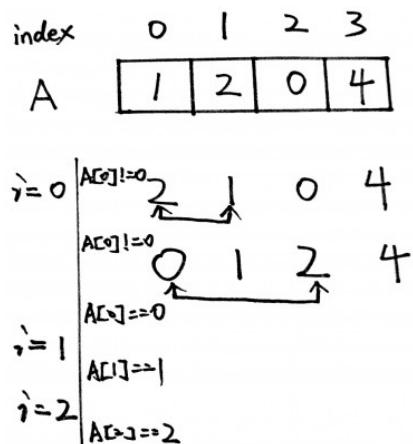
Given an unsorted integer array, find the first missing positive integer. For example, given [1,2,0] return 3 and [3,4,-1,1] return 2.

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

## 185.1 Analysis

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

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



---

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

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

```
        A[i] = A[temp];
        A[temp] = temp;
    }
}

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

return n;
}
```

---

## 185.2 Java Solution

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

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

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

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

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

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

    return n + 1;
}
```

---

# 186 Sort Colors

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

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

## 186.1 Java Solution 1 - Counting Sort

Check out this [animation](#) to understand how counting sort works.

---

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

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

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

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

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

---

## 186.2 Java Solution 2 - Improved Counting Sort

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

## 186 Sort Colors

---

```
public void sortColors(int[] nums) {  
    if(nums==null||nums.length<2){  
        return;  
    }  
  
    int[] countArray = new int[3];  
    for(int i=0; i<nums.length; i++){  
        countArray[nums[i]]++;  
    }  
  
    int j = 0;  
    int k = 0;  
    while(j<=2){  
        if(countArray[j]!=0){  
            nums[k++]=j;  
            countArray[j] = countArray[j]-1;  
        }else{  
            j++;  
        }  
    }  
}
```

---

# 187 Edit Distance in Java

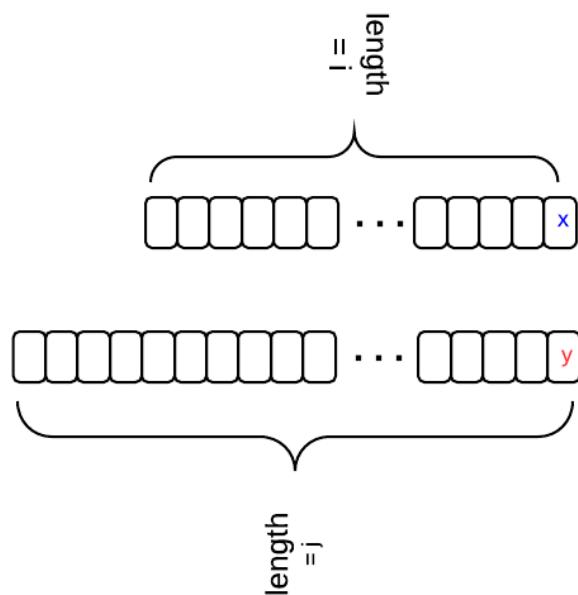
From Wiki:

*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.

## 187.1 Key Analysis

Let  $dp[i][j]$  stands for the edit distance between two strings with length  $i$  and  $j$ , i.e.,  $word1[0, \dots, i-1]$  and  $word2[0, \dots, j-1]$ . 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 its 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  $x \neq y$ , and we delete  $x$  for word1, then  $dp[i][j] = dp[i-1][j] + 1$
- if  $x \neq y$ , 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:  $dp[i][0] = i$ ,  $dp[0][j] = j$

## 187.2 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 + 1][j] + 1;  
                int delete = dp[i][j + 1] + 1;  
  
                int min = replace > insert ? insert : replace;  
                min = delete > min ? min : delete;  
                dp[i + 1][j + 1] = min;  
            }  
        }  
    }  
  
    return dp[len1][len2];  
}
```





# 188 Distinct Subsequences Total

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

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

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

Return 3.

## 188.1 Analysis

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

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

## 188.2 Java Solution 1

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

---

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

    for (int i = 0; i < S.length(); i++)
        table[i][0] = 1;

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

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

---

### 188.3 Java Solution 2

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

---

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

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

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

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

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

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

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

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

---

# 189 Longest Palindromic Substring

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

## 189.1 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  $\text{table}[i][j]$  denote whether a substring from i to j is palindrome.

Changing condition:

---

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

---

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

```
public String longestPalindrome(String s) {
    if(s==null || s.length()==0)
        return null;
    if(s.length()==1)
        return s;

    int len = s.length();

    int maxLen = 0;
    String longest = null;
    boolean [][] dp = new boolean[len][len];

    for(int i=len-1; i>=0; i--){
        for(int j=i; j<len; j++){
            if(s.charAt(i)==s.charAt(j) && (j-i<=2||dp[i+1][j-1])){
                dp[i][j]=true;
                if(j-i+1 > maxLen){
                    longest = s.substring(i, j+1);
                    maxLen = j-i+1;
                }
            }
        }
    }

    return longest;
```

```
}
```

---

For example, if the input string is "dabcba", the final matrix would be the following:

---

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

---

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

## 189.2 A Simple Algorithm

Time  $O(n^2)$ , 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);
}
```

---

### 189.3 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.



# 190 Word Break

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

## 190.1 Naive Approach

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

---

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

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

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

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

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

        return false;
    }
}
```

---

Time is  $O(n^2)$  and exceeds the time limit.

## 190.2 Dynamic Programming

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

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

---

```
public class Solution {  
    public boolean wordBreak(String s, Set<String> dict) {  
        boolean[] t = new boolean[s.length()+1];  
        t[0] = true; //set first to be true, why?  
        //Because we need initial state  
  
        for(int i=0; i<s.length(); i++){  
            //should continue from match position  
            if(!t[i])  
                continue;  
  
            for(String a: dict){  
                int len = a.length();  
                int end = i + len;  
                if(end > s.length())  
                    continue;  
  
                if(t[end]) continue;  
  
                if(s.substring(i, end).equals(a)){  
                    t[end] = true;  
                }  
            }  
        }  
  
        return t[s.length()];  
    }  
}
```

---

Time: O(string length \* dict size)

One tricky part of this solution is the case:

---

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

---

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

## 190.3 Regular Expression

The problem is equivalent to matching the regular expression (leet|code)\*, which means that it can be solved by building a DFA in  $O(2^m)$  and executing it in  $O(n)$ .

(Thanks to hdante.) Leetcode online judge does not allow using the Pattern class though.

---

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

    StringBuilder sb = new StringBuilder();

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

    String pattern = sb.toString().substring(0, sb.length()-1);
    pattern = "("+pattern+")*";
    Pattern p = Pattern.compile(pattern);
    Matcher m = p.matcher("goalspecial");

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

---

## 190.4 The More Interesting Problem

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

Check out [Word Break II](#).



# 191 Word Break II

Given a string s and a dictionary of words dict, add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences. For example, given s = "catsanddog", dict = ["cat", "cats", "and", "sand", "dog"], the solution is ["cats and dog", "cat sand dog"].

## 191.1 Java Solution - Dynamic Programming

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

The following diagram shows the structure of the tracking array.

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

---

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

## 191 Word Break II

---

```
if( dp[i] == null )
    continue;

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

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

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

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

return result;
}

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

        result.add(path);
        return;
    }

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

---

This problem is also useful for solving real problems. Assuming you want to analyze the domain names of the top 10k websites. We can use this solution to break the main part of the domain into words and then get a sense of what kinds of websites are

popular. I did this a long time ago and found some interesting results. For example, the most frequent words include "news", "tube", "porn", "etc".



# 192 Maximum Subarray

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

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

## 192.1 Wrong Solution

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

---

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

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

            if (sum < 0)
                sum = 0;
        }

        return maxSum;
    }
}
```

---

## 192.2 Dynamic Programming Solution

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

---

```
public class Solution {
    public int maxSubArray(int[] A) {
        int max = A[0];
        int[] sum = new int[A.length];
        sum[0] = A[0];

        for (int i = 1; i < A.length; i++) {
```

```
    sum[i] = Math.max(A[i], sum[i - 1] + A[i]);
    max = Math.max(max, sum[i]);
}

return max;
}
```

---

### 192.3 Simple Solution

Mehdi provided the following solution in his comment.

---

```
public int maxSubArray(int[] A) {
    int newsum=A[0];
    int max=A[0];
    for(int i=1;i<A.length;i++){
        newsum=Math.max(newsum+A[i],A[i]);
        max= Math.max(max, newsum);
    }
    return max;
}
```

---

This problem is asked by Palantir.

# 193 Maximum Product Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array [2,3,-2,4], the contiguous subarray [2,3] has the largest product = 6.

## 193.1 Java Solution 1 - Brute-force

---

```
public int maxProduct(int[] A) {
    int max = Integer.MIN_VALUE;

    for(int i=0; i<A.length; i++){
        for(int l=0; l<A.length; l++){
            if(i+l < A.length){
                int product = calProduct(A, i, l);
                max = Math.max(product, max);
            }
        }
    }
    return max;
}

public int calProduct(int[] A, int i, int j){
    int result = 1;
    for(int m=i; m<=j; m++){
        result = result * A[m];
    }
    return result;
}
```

---

The time of the solution is  $O(n^2)$ .

## 193.2 Java Solution 2 - Dynamic Programming

This is similar to [maximum subarray](#). Instead of sum, the sign of number affect the product value.

When iterating the array, each element has two possibilities: positive number or negative number. We need to track a minimum value, so that when a negative number

## 193 Maximum Product Subarray

---

is given, it can also find the maximum value. We define two local variables, one tracks the maximum and the other tracks the minimum.

```
public int maxProduct(int[] A) {  
    if(A==null || A.length==0)  
        return 0;  
  
    int maxLocal = A[0];  
    int minLocal = A[0];  
    int global = A[0];  
  
    for(int i=1; i<A.length; i++){  
        int temp = maxLocal;  
        maxLocal = Math.max(Math.max(A[i]*maxLocal, A[i]), A[i]*minLocal);  
        minLocal = Math.min(Math.min(A[i]*temp, A[i]), A[i]*minLocal);  
        global = Math.max(global, maxLocal);  
    }  
    return global;  
}
```

---

Time is O(n).

# 194 Palindrome Partitioning

## 194.1 Problem

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

For example, given s = "aab", Return

```
[  
  ["aa", "b"],  
  ["a", "a", "b"]  
]
```

## 194.2 Depth-first Search

```
public ArrayList<ArrayList<String>> partition(String s) {  
    ArrayList<ArrayList<String>> result = new ArrayList<ArrayList<String>>();  
  
    if (s == null || s.length() == 0) {  
        return result;  
    }  
  
    ArrayList<String> partition = new ArrayList<String>(); //track each possible  
    //partition  
    addPalindrome(s, 0, partition, result);  
  
    return result;  
}  
  
private void addPalindrome(String s, int start, ArrayList<String> partition,  
    ArrayList<ArrayList<String>> result) {  
    //stop condition  
    if (start == s.length()) {  
        ArrayList<String> temp = new ArrayList<String>(partition);  
        result.add(temp);  
        return;  
    }  
  
    for (int i = start + 1; i <= s.length(); i++) {  
        String str = s.substring(start, i);  
        if (isPalindrome(str)) {  
            partition.add(str);  
            addPalindrome(s, i, partition, result);  
            partition.remove(partition.size() - 1);  
        }  
    }  
}
```

```
        partition.add(str);
        addPalindrome(s, i, partition, result);
        partition.remove(partition.size() - 1);
    }
}

private boolean isPalindrome(String str) {
    int left = 0;
    int right = str.length() - 1;

    while (left < right) {
        if (str.charAt(left) != str.charAt(right)) {
            return false;
        }

        left++;
        right--;
    }

    return true;
}
```

---

### 194.3 Dynamic Programming

The dynamic programming approach is very similar to the problem of [longest palindrome substring](#).

---

```
public static List<String> palindromePartitioning(String s) {

    List<String> result = new ArrayList<String>();

    if (s == null)
        return result;

    if (s.length() <= 1) {
        result.add(s);
        return result;
    }

    int length = s.length();

    int[][] table = new int[length][length];

    // l is length, i is index of left boundary, j is index of right boundary
    for (int l = 1; l <= length; l++) {
        for (int i = 0; i <= length - l; i++) {
            int j = i + l - 1;
```

```
if (s.charAt(i) == s.charAt(j)) {
    if (l == 1 || l == 2) {
        table[i][j] = 1;
    } else {
        table[i][j] = table[i + 1][j - 1];
    }
    if (table[i][j] == 1) {
        result.add(s.substring(i, j + 1));
    }
} else {
    table[i][j] = 0;
}
}

return result;
}
```

---



# 195 Palindrome Partitioning II

Given a string  $s$ , partition  $s$  such that every substring of the partition is a palindrome. Return the minimum cuts needed for a palindrome partitioning of  $s$ . For example, given  $s = "aab"$ , return 1 since the palindrome partitioning  $["aa","b"]$  could be produced using 1 cut.

## 195.1 Analysis

This problem is similar to [Palindrome Partitioning](#). It can be efficiently solved by using dynamic programming. Unlike "Palindrome Partitioning", we need to maintain two cache arrays, one tracks the partition position and one tracks the number of minimum cut.

## 195.2 Java Solution

---

```
public int minCut(String s) {
    int n = s.length();

    boolean dp[][] = new boolean[n][n];
    int cut[] = new int[n];

    for (int j = 0; j < n; j++) {
        cut[j] = j; //set maximum # of cut
        for (int i = 0; i <= j; i++) {
            if (s.charAt(i) == s.charAt(j) && (j - i <= 1 || dp[i+1][j-1])) {
                dp[i][j] = true;

                // if need to cut, add 1 to the previous cut[i-1]
                if (i > 0){
                    cut[j] = Math.min(cut[j], cut[i-1] + 1);
                }else{
                    // if [0...j] is palindrome, no need to cut
                    cut[j] = 0;
                }
            }
        }
    }

    return cut[n-1];
}
```

---



# 196 House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

## 196.1 Java Solution 1 - Dynamic Programming

The key is to find the relation  $dp[i] = \max(dp[i-1], dp[i-2]+num[i-1])$ .

---

```
public int rob(int[] num) {
    if(num==null || num.length==0)
        return 0;

    int n = num.length;

    int[] dp = new int[n+1];
    dp[0]=0;
    dp[1]=num[0];

    for (int i=2; i<n+1; i++){
        dp[i] = Math.max(dp[i-1], dp[i-2]+num[i-1]);
    }

    return dp[n];
}
```

---

## 196.2 Java Solution 2

We can use two variables, even and odd, to track the maximum value so far as iterating the array. You can use the following example to walk through the code.

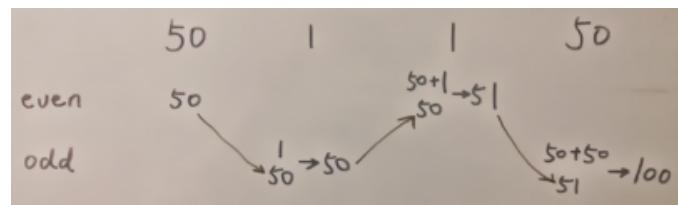
---

50 1 1 50

---

## 196 House Robber

---



---

```
public int rob(int[] num) {
    if(num==null || num.length == 0)
        return 0;

    int even = 0;
    int odd = 0;

    for (int i = 0; i < num.length; i++) {
        if (i % 2 == 0) {
            even += num[i];
            even = even > odd ? even : odd;
        } else {
            odd += num[i];
            odd = even > odd ? even : odd;
        }
    }

    return even > odd ? even : odd;
}
```

---

# 197 House Robber II

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

## 197.1 Analysis

This is an extension of [House Robber](#). There are two cases here 1) 1st element is included and last is not included 2) 1st is not included and last is included. Therefore, we can use the similar dynamic programming approach to scan the array twice and get the larger value.

## 197.2 Java Solution

---

```
public int rob(int[] nums) {
    if(nums==null||nums.length==0)
        return 0;

    int n = nums.length;

    if(n==1){
        return nums[0];
    }
    if(n==2){
        return Math.max(nums[1], nums[0]);
    }

    //include 1st element, and not last element
    int[] dp = new int[n+1];
    dp[0]=0;
    dp[1]=nums[0];

    for(int i=2; i<n; i++){
        dp[i] = Math.max(dp[i-1], dp[i-2]+nums[i-1]);
```

## 197 House Robber II

---

```
}

//not include frist element, and include last element
int[] dr = new int[n+1];
dr[0]=0;
dr[1]=nums[1];

for(int i=2; i<n; i++){
    dr[i] = Math.max(dr[i-1], dr[i-2]+nums[i]);
}

return Math.max(dp[n-1], dr[n-1]);
}
```

---

# 198 House Robber III

The houses form a binary tree. If the root is robbed, its left and right can not be robbed.

## 198.1 Analysis

Traverse down the tree recursively. We can use an array to keep 2 values: the maximum money when a root is selected and the maximum value when a root if NOT selected.

## 198.2 Java Solution

---

```
public int rob(TreeNode root) {
    if(root == null)
        return 0;

    int[] result = helper(root);
    return Math.max(result[0], result[1]);
}

public int[] helper(TreeNode root){
    if(root == null){
        int[] result = {0, 0};
        return result;
    }

    int[] result = new int[2];
    int[] left = helper(root.left);
    int[] right = helper (root.right);

    // result[0] is when root is selected, result[1] is when not.
    result[0] = root.val + left[1] + right[1];
    result[1] = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);

    return result;
}
```

---



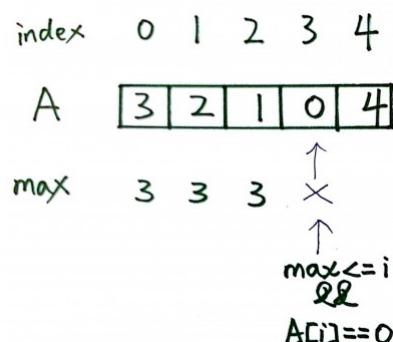
# 199 Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position. Determine if you are able to reach the last index. For example: A = [2,3,1,1,4], return true. A = [3,2,1,0,4], return false.

## 199.1 Analysis

We can track the maximum index that can be reached. The key to solve this problem is to find: 1) when the current position can not reach next position (return false), and 2) when the maximum index can reach the end (return true).

The largest index that can be reached is:  $i + A[i]$ .



## 199.2 Java Solution

---

```
public boolean canJump(int[] A) {
    if(A.length <= 1)
        return true;

    int max = A[0]; //max stands for the largest index that can be reached.

    for(int i=0; i<A.length; i++){
        //if not enough to go to next
        if(max <= i && A[i] == 0)
            return false;

        //update max
```

## 199 Jump Game

---

```
if(i + A[i] > max){  
    max = i + A[i];  
}  
  
//max is enough to reach the end  
if(max >= A.length-1)  
    return true;  
}  
  
return false;  
}
```

---

# 200 Jump Game II

Given an array of non-negative integers, you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

For example, given array A = [2,3,1,1,4], the minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

## 200.1 Analysis

This is an extension of [Jump Game](#).

The solution is similar, but we also track the maximum steps of last jump.

## 200.2 Java Solution

---

```
public int jump(int[] nums) {
    if (nums == null || nums.length == 0)
        return 0;

    int lastReach = 0;
    int reach = 0;
    int step = 0;

    for (int i = 0; i <= reach && i < nums.length; i++) {
        //when last jump can not read current i, increase the step by 1
        if (i > lastReach) {
            step++;
            lastReach = reach;
        }
        //update the maximal jump
        reach = Math.max(reach, nums[i] + i);
    }

    if (reach < nums.length - 1)
        return 0;

    return step;
}
```

---



# 201 Best Time to Buy and Sell Stock

Say you have an array for which the  $i$ th element is the price of a given stock on day  $i$ .

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

## 201.1 Naive Approach

The naive approach exceeds time limit.

---

```
public int maxProfit(int[] prices) {
    if(prices == null || prices.length < 2){
        return 0;
    }

    int profit = Integer.MIN_VALUE;
    for(int i=0; i<prices.length-1; i++){
        for(int j=i+1; j< prices.length; j++){
            if(profit < prices[j] - prices[i]){
                profit = prices[j] - prices[i];
            }
        }
    }
    return profit;
}
```

---

## 201.2 Efficient Approach

Instead of keeping track of largest element in the array, we track the maximum profit so far.

---

```
public int maxProfit(int[] prices) {
    int profit = 0;
    int minElement = Integer.MAX_VALUE;
    for(int i=0; i<prices.length; i++){
        profit = Math.max(profit, prices[i]-minElement);
        minElement = Math.min(minElement, prices[i]);
    }
    return profit;
}
```

---



## 202 Best Time to Buy and Sell Stock II

Say you have an array for which the  $i$ th element is the price of a given stock on day  $i$ .

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

### 202.1 Analysis

This problem can be viewed as finding all ascending sequences. For example, given  $5, 1, 2, 3, 4$ , buy at  $1$  & sell at  $4$  is the same as buy at  $1$  & sell at  $2$  & buy at  $2$  & sell at  $3$  & buy at  $3$  & sell at  $4$ .

We can scan the array once, and find all pairs of elements that are in ascending order.

### 202.2 Java Solution

---

```
public int maxProfit(int[] prices) {
    int profit = 0;
    for(int i=1; i<prices.length; i++){
        int diff = prices[i]-prices[i-1];
        if(diff > 0){
            profit += diff;
        }
    }
    return profit;
}
```

---



## 203 Best Time to Buy and Sell Stock III

Say you have an array for which the  $i$ th element is the price of a given stock on day  $i$ .

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note: A transaction is a buy & a sell. You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

### 203.1 Analysis

Comparing to I and II, III limits the number of transactions to 2. This can be solved by "divide and conquer". We use  $\text{left}[i]$  to track the maximum profit for transactions before  $i$ , and use  $\text{right}[i]$  to track the maximum profit for transactions after  $i$ . You can use the following example to understand the Java solution:

---

```
Prices: 1 4 5 7 6 3 2 9
left = [0, 3, 4, 6, 6, 6, 6, 8]
right= [8, 7, 7, 7, 7, 7, 7, 0]
```

---

The maximum profit = 13

### 203.2 Java Solution

---

```
public int maxProfit(int[] prices) {
    if (prices == null || prices.length < 2) {
        return 0;
    }

    //highest profit in 0 ... i
    int[] left = new int[prices.length];
    int[] right = new int[prices.length];

    // DP from left to right
    left[0] = 0;
    int min = prices[0];
    for (int i = 1; i < prices.length; i++) {
        min = Math.min(min, prices[i]);
        left[i] = Math.max(left[i - 1], prices[i] - min);
    }

    // DP from right to left
```

## 203 Best Time to Buy and Sell Stock III

---

```
right[prices.length - 1] = 0;
int max = prices[prices.length - 1];
for (int i = prices.length - 2; i >= 0; i--) {
    max = Math.max(max, prices[i]);
    right[i] = Math.max(right[i + 1], max - prices[i]);
}

int profit = 0;
for (int i = 0; i < prices.length; i++) {
    profit = Math.max(profit, left[i] + right[i]);
}

return profit;
}
```

---

# 204 Best Time to Buy and Sell Stock IV

## 204.1 Problem

Say you have an array for which the  $i$ th element is the price of a given stock on day  $i$ . Design an algorithm to find the maximum profit. You may complete at most  $k$  transactions.

Note: You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

## 204.2 Analysis

This is a generalized version of [Best Time to Buy and Sell Stock III](#). If we can solve this problem, we can also use  $k=2$  to solve III.

The problem can be solved by using dynamic programming. The relation is:

---

```
local[i][j] = max(global[i-1][j-1] + max(diff, 0), local[i-1][j]+diff)
global[i][j] = max(local[i][j], global[i-1][j])
```

---

We track two arrays - local and global. The local array tracks maximum profit of  $j$  transactions & the last transaction is on  $i$ th day. The global array tracks the maximum profit of  $j$  transactions until  $i$ th day.

## 204.3 Java Solution - 2D Dynamic Programming

---

```
public int maxProfit(int k, int[] prices) {
    int len = prices.length;

    if (len < 2 || k <= 0)
        return 0;

    // ignore this line
    if (k == 1000000000)
        return 1648961;

    int[][] local = new int[len][k + 1];
    int[][] global = new int[len][k + 1];

    for (int i = 1; i < len; i++) {
        int diff = prices[i] - prices[i - 1];
```

```
        for (int j = 1; j <= k; j++) {
            local[i][j] = Math.max(
                global[i - 1][j - 1] + Math.max(diff, 0),
                local[i - 1][j] + diff);
            global[i][j] = Math.max(global[i - 1][j], local[i][j]);
        }
    }

    return global[prices.length - 1][k];
}
```

---

## 204.4 Java Solution - 1D Dynamic Programming

The solution above can be simplified to be the following:

---

```
public int maxProfit(int k, int[] prices) {
    if (prices.length < 2 || k <= 0)
        return 0;

    //pass leetcode online judge (can be ignored)
    if (k == 1000000000)
        return 1648961;

    int[] local = new int[k + 1];
    int[] global = new int[k + 1];

    for (int i = 0; i < prices.length - 1; i++) {
        int diff = prices[i + 1] - prices[i];
        for (int j = k; j >= 1; j--) {
            local[j] = Math.max(global[j - 1] + Math.max(diff, 0), local[j] + diff);
            global[j] = Math.max(local[j], global[j]);
        }
    }

    return global[k];
}
```

---

# 205 Dungeon Game

Example:

---

```
-2 (K) -3 3
-5 -10 1
10 30 -5 (P)
```

---

## 205.1 Java Solution

This problem can be solved by using dynamic programming. We maintain a 2-D table.  $h[i][j]$  is the minimum health value before he enters  $(i,j)$ .  $h[0][0]$  is the value of the answer. The left part is filling in numbers to the table.

```
public int calculateMinimumHP(int[][][] dungeon) {
    int m = dungeon.length;
    int n = dungeon[0].length;

    //init dp table
    int[][] h = new int[m][n];

    h[m - 1][n - 1] = Math.max(1 - dungeon[m - 1][n - 1], 1);

    //init last row
    for (int i = m - 2; i >= 0; i--) {
        h[i][n - 1] = Math.max(h[i + 1][n - 1] - dungeon[i][n - 1], 1);
    }

    //init last column
    for (int j = n - 2; j >= 0; j--) {
        h[m - 1][j] = Math.max(h[m - 1][j + 1] - dungeon[m - 1][j], 1);
    }

    //calculate dp table
    for (int i = m - 2; i >= 0; i--) {
        for (int j = n - 2; j >= 0; j--) {
            int down = Math.max(h[i + 1][j] - dungeon[i][j], 1);
            int right = Math.max(h[i][j + 1] - dungeon[i][j], 1);
            h[i][j] = Math.min(right, down);
        }
    }

    return h[0][0];
}
```

## 205 Dungeon Game

---

}

---

# 206 Decode Ways

A message containing letters from A-Z is being encoded to numbers using the following mapping:

'A' -> 1 'B' -> 2 ... 'Z' -> 26

Given an encoded message containing digits, determine the total number of ways to decode it.

## 206.1 Analysis

## 206.2 Java Solution

---

```
public int numDecodings(String s) {
    if(s==null||s.length()==0||s.equals("0"))
        return 0;

    int[] t = new int[s.length()+1];
    t[0] = 1;

    //if(s.charAt(0)!='0')
    if(isValid(s.substring(0,1)))
        t[1]=1;
    else
        t[1]=0;

    for(int i=2; i<=s.length(); i++){
        if(isValid(s.substring(i-1,i))){
            t[i]+=t[i-1];
        }

        if(isValid(s.substring(i-2,i))){
            t[i]+=t[i-2];
        }
    }

    return t[s.length()];
}

public boolean isValid(String s){
    if(s.charAt(0)=='0')
        return false;
```

## 206 Decode Ways

---

```
int value = Integer.parseInt(s);
return value>=1&&value<=26;
}
```

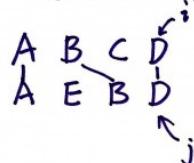
---

## 207 Longest Common Subsequence

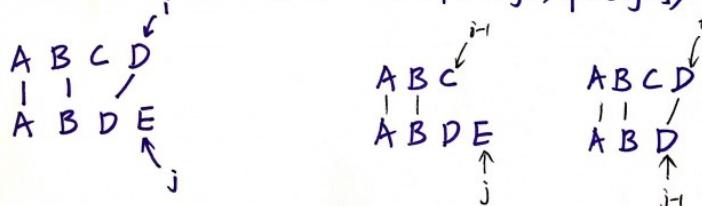
The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences).

### 207.1 Analysis

if  $a[i] == b[j]$ , then  $dp[i+1][j+1] = dp[i][j] + 1$



if  $a[i] \neq b[j]$ , then  $dp[i+1][j+1] = \max(dp[i+1][j], dp[i][j+1])$



### 207.2 Java Solution

```
public static int getLongestCommonSubsequence(String a, String b){  
    int m = a.length();  
    int n = b.length();  
    int[][] dp = new int[m+1][n+1];  
  
    for(int i=0; i<=m; i++){  
        for(int j=0; j<=n; j++){  
            if(i==0 || j==0){  
                dp[i][j]=0;  
            }else if(a.charAt(i-1)==b.charAt(j-1)){  
                dp[i][j] = 1 + dp[i-1][j-1];  
            }else{  
                dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);  
            }  
        }  
    }  
}
```

## 207 Longest Common Subsequence

---

```
    }  
  
    return dp[m][n];  
}
```

---

## 208 Longest Common Substring

In computer science, the longest common substring problem is to find the longest string that is a substring of two or more strings.

### 208.1 Analysis

Given two strings  $a$  and  $b$ , let  $dp[i][j]$  be the length of the common substring ending at  $a[i]$  and  $b[j]$ .

$$\begin{array}{c} i \\ \downarrow \\ a b c \quad \text{when } i=0 \text{ or } j=0 \\ | \\ a b c d \quad dp[i][j] = 1 \\ \uparrow \\ j \end{array}$$
$$\begin{array}{c} i \\ \downarrow \\ a b c \quad \text{when } i>0 \text{ and } j>0 \\ | \\ a b c d \quad dp[i][j] = dp[i-1][j-1] + 1 \\ | \\ = dp[0][0] + 1 \\ = 1 + 1 \\ = 2 \end{array}$$

The dp table looks like the following given  $a = "abc"$  and  $b = "abcd"$ .

	a	b	c
a	1	0	0
b	0	2	0
c	0	0	3
d	0	0	0

### 208.2 Java Solution

```
public static int getLongestCommonSubstring(String a, String b){
    int m = a.length();
    int n = b.length();

    int max = 0;

    int[][] dp = new int[m][n];

    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(a.charAt(i) == b.charAt(j)){
                if(i==0 || j==0){
                    dp[i][j]=1;
                }else{
                    dp[i][j] = dp[i-1][j-1]+1;
                }

                if(max < dp[i][j])
                    max = dp[i][j];
            }
        }
    }

    return max;
}
```

---

This is a similar problem like [longest common subsequence](#). The difference of the solution is that for this problem when  $a[i] \neq b[j]$ ,  $dp[i][j]$  are all zeros by default. However, in the [longest common subsequence](#) problem,  $dp[i][j]$  values are carried from the previous values, i.e.,  $dp[i-1][j]$  and  $dp[i][j-1]$ .

# 209 Longest Increasing Subsequence

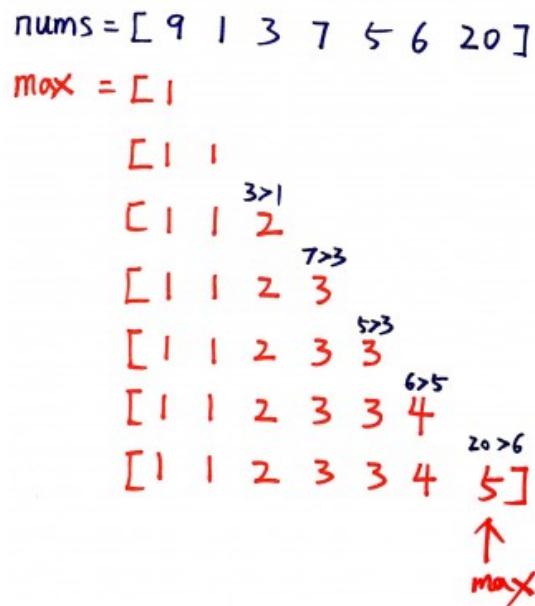
Given an unsorted array of integers, find the length of longest increasing subsequence.

For example, given [10, 9, 2, 5, 3, 7, 101, 18], the longest increasing subsequence is [2, 3, 7, 101]. Therefore the length is 4.

## 209.1 Java Solution 1 - Dynamic Programming

Let  $\text{max}[i]$  represent the length of the longest increasing subsequence so far. If any element before  $i$  is smaller than  $\text{nums}[i]$ , then  $\text{max}[i] = \max(\text{max}[i], \text{max}[j]+1)$ .

Here is an example:



---

```
public int lengthOfLIS(int[] nums) {  
    if(nums==null || nums.length==0)  
        return 0;  
  
    int[] max = new int[nums.length];  
  
    for(int i=0; i<nums.length; i++){  
        max[i]=1;
```

```
        for(int j=0; j<i;j++){  
            if(nums[i]>nums[j]){  
                max[i]=Math.max(max[i], max[j]+1);  
            }  
        }  
  
        int result = 0;  
        for(int i=0; i<max.length; i++){  
            if(max[i]>result)  
                result = max[i];  
        }  
        return result;  
    }

---


```

## 209.2 Java Solution 2 - O(nlog(n))

We can put the increasing sequence in a list.

---

```
for each num in nums  
    if(list.size()==0)  
        add num to list  
    else if(num > last element in list)  
        add num to list  
    else  
        replace the element in the list which is the smallest but bigger than  
        num

---


```

nums = [ 9 1 3 7 5 6 20 ]

9  
|  
| 3  
| 3 7  
| 3 5  
| 3 5 6  
| 3 5 6 20

---

```
public int lengthOfLIS(int[] nums) {  
    if(nums==null || nums.length==0)
```

```
return 0;

ArrayList<Integer> list = new ArrayList<Integer>();

for(int num: nums){
    if(list.size()==0){
        list.add(num);
    }else if(num>list.get(list.size()-1)){
        list.add(num);
    }else{
        int i=0;
        int j=list.size()-1;

        while(i<j){
            int mid = (i+j)/2;
            if(list.get(mid) < num){
                i=mid+1;
            }else{
                j=mid;
            }
        }

        list.set(j, num);
    }
}

return list.size();
}
```

---



## 210 Coin Change

You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

### 210.1 Java Solution 1 - Dynamic Programming

---

Let  $dp[v]$  to be the minimum number of coins required to get the amount  $v$ .  
 $dp[i+a\_coin] = \min(dp[i+a\_coin], dp[i]+1)$  if  $dp[i]$  is reachable.  
 $dp[i+a\_coin] = dp[i+a\_coin]$  if  $dp[i]$  is not reachable.  
We initially set  $dp[i]$  to be MAX\_VALUE.

---

Here is the Java code:

---

```
public int coinChange(int[] coins, int amount) {
    if(amount==0) return 0;

    int[] dp = new int [amount+1];
    dp[0]=0; // do not need any coin to get 0 amount
    for(int i=1;i<=amount; i++)
        dp[i]= Integer.MAX_VALUE;

    for(int i=0; i<=amount; i++){
        for(int coin: coins){
            if(i+coin <=amount){
                if(dp[i]==Integer.MAX_VALUE){
                    dp[i+coin] = dp[i+coin];
                }else{
                    dp[i+coin] = Math.min(dp[i+coin], dp[i]+1);
                }
            }
        }
    }

    if(dp[amount] >= Integer.MAX_VALUE)
        return -1;

    return dp[amount];
}
```

---

This clean solution takes time  $O(n^2)$ .

## 210.2 Java Solution 2 - Breath First Search (BFS)

Most dynamic programming problems can be solved by using BFS.

We can view this problem as going to a target position with steps that are allows in the array coins. We maintain two queues: one of the amount so far and the other for the minimal steps. The time is too much because of the contains method take n and total time is  $O(n^3)$ .

---

```
public int coinChange(int[] coins, int amount) {
    if (amount == 0)
        return 0;

    LinkedList<Integer> amountQueue = new LinkedList<Integer>();
    LinkedList<Integer> stepQueue = new LinkedList<Integer>();

    // to get 0, 0 step is required
    amountQueue.offer(0);
    stepQueue.offer(0);

    while (amountQueue.size() > 0) {
        int temp = amountQueue.poll();
        int step = stepQueue.poll();

        if (temp == amount)
            return step;

        for (int coin : coins) {
            if (temp > amount) {
                continue;
            } else {
                if (!amountQueue.contains(temp + coin)) {
                    amountQueue.offer(temp + coin);
                    stepQueue.offer(step + 1);
                }
            }
        }
    }

    return -1;
}
```

---

## 211 Perfect Squares

Given a positive integer  $n$ , find the least number of perfect square numbers (for example, 1, 4, 9, 16, ...) which sum to  $n$ .

For example, given  $n = 12$ , return 3 because  $12 = 4 + 4 + 4$ ; given  $n = 13$ , return 2 because  $13 = 4 + 9$ .

### 211.1 Java Solution

This is a dp problem. The key is to find the relation which is  $dp[i] = \min(dp[i], dp[i-square]+1)$ . For example,  $dp[5]=dp[4]+1=1+1=2$ .

---

```
public int numSquares(int n) {
    int max = (int) Math.sqrt(n);

    int[] dp = new int[n+1];
    Arrays.fill(dp, Integer.MAX_VALUE);

    for(int i=1; i<=n; i++){
        for(int j=1; j<=max; j++){
            if(i==j*j){
                dp[i]=1;
            }else if(i>j*j){
                dp[i]=Math.min(dp[i], dp[i-j*j] + 1);
            }
        }
    }

    return dp[n];
}
```

---



## 212 Single Number

The problem:

*Given an array of integers, every element appears twice except for one. Find that single one.*

### 212.1 Java Solution 1

The key to solve this problem is bit manipulation. XOR will return 1 only on two different bits. So if two numbers are the same, XOR will return 0. Finally only one number left.

---

```
public int singleNumber(int[] A) {
    int x = 0;
    for (int a : A) {
        x = x ^ a;
    }
    return x;
}
```

---

### 212.2 Java Solution 2

---

```
public int singleNumber(int[] A) {
    HashSet<Integer> set = new HashSet<Integer>();
    for (int n : A) {
        if (!set.add(n))
            set.remove(n);
    }
    Iterator<Integer> it = set.iterator();
    return it.next();
}
```

---

The question now is do you know any other ways to do this?



# 213 Single Number II

## 213.1 Problem

Given an array of integers, every element appears three times except for one. Find that single one.

## 213.2 Java Solution

This problem is similar to [Single Number](#).

---

```
public int singleNumber(int[] A) {
    int ones = 0, twos = 0, threes = 0;
    for (int i = 0; i < A.length; i++) {
        twos |= ones & A[i];
        ones ^= A[i];
        threes = ones & twos;
        ones &= ~threes;
        twos &= ~threes;
    }
    return ones;
}
```

---



## 214 Twitter Codility Problem Max Binary Gap

Problem: Get maximum binary Gap.

For example, 9's binary form is 1001, the gap is 2.

### 214.1 Java Solution 1

An integer  $x \& 1$  will get the last digit of the integer.

---

```
public static int getGap(int N) {
    int max = 0;
    int count = -1;
    int r = 0;

    while (N > 0) {
        // get right most bit & shift right
        r = N & 1;
        N = N >> 1;

        if (0 == r && count >= 0) {
            count++;
        }

        if (1 == r) {
            max = count > max ? count : max;
            count = 0;
        }
    }

    return max;
}
```

---

Time is  $O(n)$ .

### 214.2 Java Solution 2

---

```
public static int getGap(int N) {
    int pre = -1;
    int len = 0;
```

## 214 Twitter Codility Problem Max Binary Gap

---

```
while (N > 0) {  
    int k = N & -N;  
  
    int curr = (int) Math.log(k);  
  
    N = N & (N - 1);  
  
    if (pre != -1 && Math.abs(curr - pre) > len) {  
        len = Math.abs(curr - pre) + 1;  
    }  
    pre = curr;  
}  
  
return len;  
}
```

---

Time is  $O(\log(n))$ .

# 215 Number of 1 Bits

## 215.1 Problem

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

For example, the 32-bit integer '11' has binary representation 000000000000000000000000001011, so the function should return 3.

## 215.2 Java Solution

---

```
public int hammingWeight(int n) {
    int count = 0;
    for(int i=1; i<33; i++){
        if(getBit(n, i) == true){
            count++;
        }
    }
    return count;
}

public boolean getBit(int n, int i){
    return (n & (1 << i)) != 0;
}
```

---



# 216 Reverse Bits

## 216.1 Problem

Reverse bits of a given 32 bits unsigned integer.

For example, given input 43261596 (represented in binary as 00000010100101000001111010011100), return 964176192 (represented in binary as 00111001011110000010100101000000).

Follow up: If this function is called many times, how would you optimize it?

Related problem: Reverse Integer

## 216.2 Java Solution

---

```
public int reverseBits(int n) {
    for (int i = 0; i < 16; i++) {
        n = swapBits(n, i, 32 - i - 1);
    }

    return n;
}

public int swapBits(int n, int i, int j) {
    int a = (n >> i) & 1;
    int b = (n >> j) & 1;

    if ((a ^ b) != 0) {
        n ^= (1 << i) | (1 << j);
    }

    return n;
}
```

---



# 217 Repeated DNA Sequences

## 217.1 Problem

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

For example, given s = "AAAAACCCCCAAAAACCCCCAAAAAGGGTTT", return: ["AAAAACCCCC", "CCCCAAAAAA"].

## 217.2 Java Solution

The key to solve this problem is that each of the 4 nucleotides can be stored in 2 bits. So the 10-letter-long sequence can be converted to 20-bits-long integer. The following is a Java solution. You may use an example to manually execute the program and see how it works.

```
public List<String> findRepeatedDnaSequences(String s) {
    List<String> result = new ArrayList<String>();

    int len = s.length();
    if (len < 10) {
        return result;
    }

    Map<Character, Integer> map = new HashMap<Character, Integer>();
    map.put('A', 0);
    map.put('C', 1);
    map.put('G', 2);
    map.put('T', 3);

    Set<Integer> temp = new HashSet<Integer>();
    Set<Integer> added = new HashSet<Integer>();

    int hash = 0;
    for (int i = 0; i < len; i++) {
        if (i < 9) {
            //each ACGT fit 2 bits, so left shift 2
            hash = (hash << 2) + map.get(s.charAt(i));
        } else {
```

## 217 Repeated DNA Sequences

---

```
hash = (hash << 2) + map.get(s.charAt(i));
//make length of hash to be 20
hash = hash & (1 << 20) - 1;

if (temp.contains(hash) && !added.contains(hash)) {
    result.add(s.substring(i - 9, i + 1));
    added.add(hash); //track added
} else {
    temp.add(hash);
}
}

return result;
}
```

---

## 218 Bitwise AND of Numbers Range

**218.1 Given a range [m, n] where  $0 \leq m \leq n \leq 2147483647$ , return the bitwise AND of all numbers in this range, inclusive. For example, given the range [5, 7], you should return 4. Java Solution**

The key to solve this problem is bitwise AND consecutive numbers. You can use the following example to walk through the code.

---

```
8 4 2 1
-----
5 | 0 1 0 1
6 | 0 1 1 0
7 | 0 1 1 1
```

---

---

```
public int rangeBitwiseAnd(int m, int n) {
    while (n > m) {
        n = n & n - 1;
    }
    return m & n;
}
```

---



## 219 Power of Two

Given an integer, write a function to determine if it is a power of two.

### 219.1 Analysis

If a number is power of 2, its binary form should be 10...0. So if we right shift a bit of the number and then left shift a bit, the value should be the same when the number  $\geq 10$ (i.e.,2).

### 219.2 Java Solution

---

```
public boolean isPowerOfTwo(int n) {
    if(n<=0)
        return false;

    while(n>2){
        int t = n>>1;
        int c = t<<1;

        if(n-c != 0)
            return false;

        n = n>>1;
    }

    return true;
}
```

---



## 220 Counting Bits

Given a non negative integer number num. For every numbers i in the range  $0 \leq i \leq$  num calculate the number of 1's in their binary representation and return them as an array.

Example:

For num = 5 you should return [0,1,1,2,1,2].

### 220.1 Naive Solution

We can simply count bits for each number like the following:

---

```
public int[] countBits(int num) {
    int[] result = new int[num+1];

    for(int i=0; i<=num; i++){
        result[i] = countEach(i);
    }

    return result;
}

public int countEach(int num){
    int result = 0;

    while(num!=0){
        if(num%2==1){
            result++;
        }
        num = num/2;
    }

    return result;
}
```

---

### 220.2 Improved Solution

For number  $2(10)$ ,  $4(100)$ ,  $8(1000)$ ,  $16(10000)$ , ..., the number of 1's is 1. Any other number can be converted to be  $2^m + x$ . For example,  $9=8+1$ ,  $10=8+2$ . The number of 1's for any other number is  $1 + \#$  of 1's in  $x$ .

Number	# of 1's
1	1
2	1
3 = 2+1	2
4	1
5 = 4+1	2
6 = 4+2	2
7 = 4+3	3 = 2+1
8	1
9 = 8+1	2
10 = 8+2	2
:	:

---

```

public int[] countBits(int num) {
    int[] result = new int[num+1];

    int p = 1; //p tracks the index for number x
    int pow = 1;
    for(int i=1; i<=num; i++){
        if(i==pow){
            result[i] = 1;
            pow <=> 1;
            p = 1;
        }else{
            result[i] = result[p]+1;
            p++;
        }
    }

    return result;
}

```

---

## 221 Maximum Product of Word Lengths

Given a string array words, find the maximum value of length(word[i]) \* length(word[j]) where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

### 221.1 Java Solution

---

```
public int maxProduct(String[] words) {
    if(words==null || words.length==0)
        return 0;

    int[] arr = new int[words.length];
    for(int i=0; i<words.length; i++){
        for(int j=0; j<words[i].length(); j++){
            char c = words[i].charAt(j);
            arr[i] |= (1<< (c-'a'));
        }
    }

    int result = 0;

    for(int i=0; i<words.length; i++){
        for(int j=i+1; j<words.length; j++){
            if((arr[i] & arr[j]) == 0){
                result = Math.max(result, words[i].length()*words[j].length());
            }
        }
    }

    return result;
}
```

---



## 222 Permutations

Given a collection of numbers, return all possible permutations.

---

For example,  
[1,2,3] have the following permutations:  
[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1].

---

### 222.1 Java Solution 1

We can get all permutations by the following steps:

---

```
[1]
[2, 1]
[1, 2]
[3, 2, 1]
[2, 3, 1]
[2, 1, 3]
[3, 1, 2]
[1, 3, 2]
[1, 2, 3]
```

---

Loop through the array, in each iteration, a new number is added to different locations of results of previous iteration. Start from an empty List.

---

```
public ArrayList<ArrayList<Integer>> permute(int[] num) {
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

    //start from an empty list
    result.add(new ArrayList<Integer>());

    for (int i = 0; i < num.length; i++) {
        //list of list in current iteration of the array num
        ArrayList<ArrayList<Integer>> current = new
            ArrayList<ArrayList<Integer>>();

        for (ArrayList<Integer> l : result) {
            // # of locations to insert is largest index + 1
            for (int j = 0; j < l.size()+1; j++) {
                // + add num[i] to different locations
                l.add(j, num[i]);

                ArrayList<Integer> temp = new ArrayList<Integer>(l);
                current.add(temp);
            }
        }
    }
}
```

---

```
        current.add(temp);

        //System.out.println(temp);

        // - remove num[i] add
        l.remove(j);
    }
}

result = new ArrayList<ArrayList<Integer>>(current);
}

return result;
}
```

---

## 222.2 Java Solution 2

We can also recursively solve this problem. Swap each element with each element after it.

```
public ArrayList<ArrayList<Integer>> permute(int[] num) {
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
    permute(num, 0, result);
    return result;
}

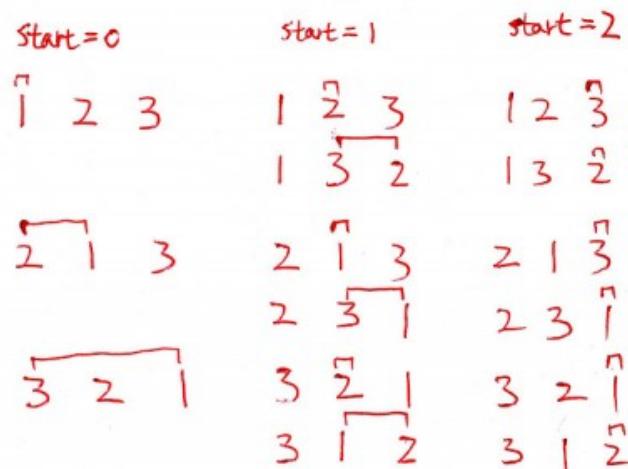
void permute(int[] num, int start, ArrayList<ArrayList<Integer>> result) {

    if (start >= num.length) {
        ArrayList<Integer> item = convertArrayToList(num);
        result.add(item);
    }

    for (int j = start; j <= num.length - 1; j++) {
        swap(num, start, j);
        permute(num, start + 1, result);
        swap(num, start, j);
    }
}

private ArrayList<Integer> convertArrayToList(int[] num) {
    ArrayList<Integer> item = new ArrayList<Integer>();
    for (int h = 0; h < num.length; h++) {
        item.add(num[h]);
    }
    return item;
}
```

```
private void swap(int[] a, int i, int j) {  
    int temp = a[i];  
    a[i] = a[j];  
    a[j] = temp;  
}
```





# 223 Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

---

For example, [1,1,2] have the following unique permutations:  
[1,1,2], [1,2,1], and [2,1,1].

---

## 223.1 Basic Idea

For each number in the array, swap it with every element after it. To avoid duplicate, we need to check the existing sequence first.

## 223.2 Java Solution 1

---

```
public ArrayList<ArrayList<Integer>> permuteUnique(int[] num) {
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
    permuteUnique(num, 0, result);
    return result;
}

private void permuteUnique(int[] num, int start,
    ArrayList<ArrayList<Integer>> result) {

    if (start >= num.length) {
        ArrayList<Integer> item = convertArrayToList(num);
        result.add(item);
    }

    for (int j = start; j <= num.length-1; j++) {
        if (containsDuplicate(num, start, j)) {
            swap(num, start, j);
            permuteUnique(num, start + 1, result);
            swap(num, start, j);
        }
    }
}

private ArrayList<Integer> convertArrayToList(int[] num) {
    ArrayList<Integer> item = new ArrayList<Integer>();
    for (int h = 0; h < num.length; h++) {
```

```
        item.add(num[h]);
    }
    return item;
}

private boolean containsDuplicate(int[] arr, int start, int end) {
    for (int i = start; i <= end-1; i++) {
        if (arr[i] == arr[end]) {
            return false;
        }
    }
    return true;
}

private void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

---

### 223.3 Java Solution 2

Use set to maintain uniqueness:

```
public static ArrayList<ArrayList<Integer>> permuteUnique(int[] num) {
    ArrayList<ArrayList<Integer>> returnList = new
        ArrayList<ArrayList<Integer>>();
    returnList.add(new ArrayList<Integer>());

    for (int i = 0; i < num.length; i++) {
        Set<ArrayList<Integer>> currentSet = new HashSet<ArrayList<Integer>>();
        for (List<Integer> l : returnList) {
            for (int j = 0; j < l.size() + 1; j++) {
                l.add(j, num[i]);
                ArrayList<Integer> T = new ArrayList<Integer>(l);
                l.remove(j);
                currentSet.add(T);
            }
        }
        returnList = new ArrayList<ArrayList<Integer>>(currentSet);
    }

    return returnList;
}
```

---

Thanks to Milan for such a simple solution!

## 224 Permutation Sequence

The set  $[1,2,3,\dots,n]$  contains a total of  $n!$  unique permutations.

By listing and labeling all of the permutations in order, We get the following sequence (ie, for  $n = 3$ ):

---

```
"123"  
"132"  
"213"  
"231"  
"312"  
"321"
```

---

Given  $n$  and  $k$ , return the  $k$ th permutation sequence. (Note: Given  $n$  will be between 1 and 9 inclusive.)

### 224.1 Java Solution 1

---

```
public class Solution {  
    public String getPermutation(int n, int k) {  
  
        // initialize all numbers  
        ArrayList<Integer> numberList = new ArrayList<Integer>();  
        for (int i = 1; i <= n; i++) {  
            numberList.add(i);  
        }  
  
        // change k to be index  
        k--;  
  
        // set factorial of n  
        int mod = 1;  
        for (int i = 1; i <= n; i++) {  
            mod = mod * i;  
        }  
  
        String result = "";  
  
        // find sequence  
        for (int i = 0; i < n; i++) {  
            mod = mod / (n - i);  
            // find the right number(curIndex) of  
            int curIndex = k / mod;
```

```
// update k
k = k % mod;

// get number according to curIndex
result += numberList.get(curIndex);
// remove from list
numberList.remove(curIndex);
}

return result.toString();
}
}
```

---

## 224.2 Java Solution 2

---

```
public class Solution {
    public String getPermutation(int n, int k) {
        boolean[] output = new boolean[n];
        StringBuilder buf = new StringBuilder("");

        int[] res = new int[n];
        res[0] = 1;

        for (int i = 1; i < n; i++)
            res[i] = res[i - 1] * i;

        for (int i = n - 1; i >= 0; i--) {
            int s = 1;

            while (k > res[i]) {
                s++;
                k = k - res[i];
            }

            for (int j = 0; j < n; j++) {
                if (j + 1 <= s && output[j]) {
                    s++;
                }
            }

            output[s - 1] = true;
            buf.append(Integer.toString(s));
        }

        return buf.toString();
    }
}
```

---

## 225 Generate Parentheses

Given  $n$  pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given  $n = 3$ , a solution set is:

---

"((()))", "(()())", "(())()", "()(())", "()()()"

---

### 225.1 Java Solution 1 - DFS

This solution is simple and clear.

```
public List<String> generateParenthesis(int n) {
    ArrayList<String> result = new ArrayList<String>();
    dfs(result, "", n, n);
    return result;
}
/*
left and right represents the remaining number of ( and ) that need to be
added.
When left > right, there are more ")" placed than "(".
Such cases are wrong
and the method stops.
*/
public void dfs(ArrayList<String> result, String s, int left, int right){
    if(left > right)
        return;

    if(left==0&&right==0){
        result.add(s);
        return;
    }

    if(left>0){
        dfs(result, s+"(", left-1, right);
    }

    if(right>0){
        dfs(result, s+")", left, right-1);
    }
}
```

---

## 225.2 Java Solution 2

This solution looks more complicated. , You can use n=2 to walk though the code.

---

```
public List<String> generateParenthesis(int n) {
    ArrayList<String> result = new ArrayList<String>();
    ArrayList<Integer> diff = new ArrayList<Integer>();

    result.add("");
    diff.add(0);

    for (int i = 0; i < 2 * n; i++) {
        ArrayList<String> temp1 = new ArrayList<String>();
        ArrayList<Integer> temp2 = new ArrayList<Integer>();

        for (int j = 0; j < result.size(); j++) {
            String s = result.get(j);
            int k = diff.get(j);

            if (i < 2 * n - 1) {
                temp1.add(s + "(");
                temp2.add(k + 1);
            }

            if (k > 0 && i < 2 * n - 1 || k == 1 && i == 2 * n - 1) {
                temp1.add(s + ")");
                temp2.add(k - 1);
            }
        }

        result = new ArrayList<String>(temp1);
        diff = new ArrayList<Integer>(temp2);
    }

    return result;
}
```

---

## 226 Combination Sum

Given a set of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T. The same repeated number may be chosen from C unlimited number of times.

Note: All numbers (including target) will be positive integers. Elements in a combination ( $a_1, a_2, \dots, a_k$ ) must be in non-descending order. (ie,  $a_1 \leq a_2 \leq \dots \leq a_k$ ). The solution set must not contain duplicate combinations. For example, given candidate set  $2,3,6,7$  and target  $7$ , A solution set is:

---

[7]

[2, 2, 3]

---

### 226.1 Thoughts

The first impression of this problem should be depth-first search(DFS). To solve DFS problem, recursion is a normal implementation.

Note that the candidates array is not sorted, we need to sort it first.

### 226.2 Java Solution

---

```
public ArrayList<ArrayList<Integer>> combinationSum(int[] candidates, int target) {
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();

    if(candidates == null || candidates.length == 0) return result;

    ArrayList<Integer> current = new ArrayList<Integer>();
    Arrays.sort(candidates);

    combinationSum(candidates, target, 0, current, result);

    return result;
}

public void combinationSum(int[] candidates, int target, int j,
    ArrayList<Integer> curr, ArrayList<ArrayList<Integer>> result){
    if(target == 0){
        ArrayList<Integer> temp = new ArrayList<Integer>(curr);
        result.add(temp);
    }
}
```

## 226 Combination Sum

---

```
        return;
    }

    for(int i=j; i<candidates.length; i++){
        if(target < candidates[i])
            return;

        curr.add(candidates[i]);
        combinationSum(candidates, target - candidates[i], i, curr, result);
        curr.remove(curr.size()-1);
    }
}
```

---

## 227 Combination Sum II

Given a collection of candidate numbers (C) and a target number (T), find all unique combinations in C where the candidate numbers sums to T. Each number in C may only be used ONCE in the combination.

Note: 1) All numbers (including target) will be positive integers. 2) Elements in a combination ( $a_1, a_2, \dots, a_k$ ) must be in non-descending order. (ie,  $a_1 \leq a_2 \leq \dots \leq a_k$ ). 3) The solution set must not contain duplicate combinations.

### 227.1 Java Solution

This problem is an extension of [Combination Sum](#). The difference is one number in the array can only be used ONCE.

---

```
public List<ArrayList<Integer>> combinationSum2(int[] num, int target) {
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
    if(num == null || num.length == 0)
        return result;

    Arrays.sort(num);

    ArrayList<Integer> temp = new ArrayList<Integer>();
    getCombination(num, 0, target, temp, result);

    HashSet<ArrayList<Integer>> set = new HashSet<ArrayList<Integer>>(result);

    //remove duplicate lists
    result.clear();
    result.addAll(set);

    return result;
}

public void getCombination(int[] num, int start, int target,
    ArrayList<Integer> temp, ArrayList<ArrayList<Integer>> result){
    if(target == 0){
        ArrayList<Integer> t = new ArrayList<Integer>(temp);
        result.add(t);
        return;
    }

    for(int i=start; i<num.length; i++){
        if(target < num[i])
```

## 227 Combination Sum II

---

```
        continue;

        temp.add(num[i]);
        getCombination(num, i+1, target-num[i], temp, result);
        temp.remove(temp.size()-1);
    }
}
```

---

## 228 Combination Sum III

Find all possible combinations of  $k$  numbers that add up to a number  $n$ , given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Ensure that numbers within the set are sorted in ascending order.

Example 1: Input:  $k = 3$ ,  $n = 7$  Output:  $[[1,2,4]]$  Example 2: Input:  $k = 3$ ,  $n = 9$  Output:  $[[1,2,6], [1,3,5], [2,3,4]]$

### 228.1 Analysis

Related problems: [Combination Sum](#), [Combination Sum II](#).

### 228.2 Java Solution

---

```
public List<List<Integer>> combinationSum3(int k, int n) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    List<Integer> list = new ArrayList<Integer>();
    dfs(result, 1, n, list, k);
    return result;
}

public void dfs(List<List<Integer>> result, int start, int sum, List<Integer>
    list, int k){
    if(sum==0 && list.size()==k){
        List<Integer> temp = new ArrayList<Integer>();
        temp.addAll(list);
        result.add(temp);
    }

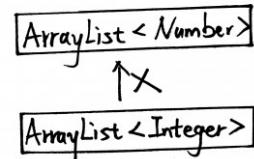
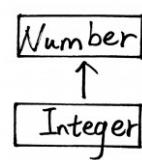
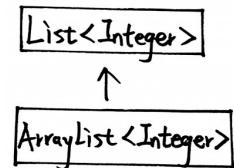
    for(int i=start; i<=9; i++){
        if(sum-i<0) break;
        if(list.size()>k) break;

        list.add(i);
        dfs(result, i+1, sum-i, list, k);
        list.remove(list.size()-1);
    }
}
```

---

Note the following relation in Java:

Integer is a subclass of Number and ArrayList is a subclass of List. But ArrayList is not a subclass of ArrayList.



# 229 Combinations

## 229.1 Problem

Given two integers n and k, return all possible combinations of k numbers out of 1 ... n.

For example, if n = 4 and k = 2, a solution is:

---

```
[  
    [2,4],  
    [3,4],  
    [2,3],  
    [1,2],  
    [1,3],  
    [1,4],  
]
```

---

## 229.2 Java Solution 1 (Recursion)

This is my naive solution. It passed the online judge. I first initialize a list with only one element, and then recursively add available elements to it.

---

```
public ArrayList<ArrayList<Integer>> combine(int n, int k) {  
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();  
  
    //illegal case  
    if (k > n) {  
        return null;  
    //if k==n  
    } else if (k == n) {  
        ArrayList<Integer> temp = new ArrayList<Integer>();  
        for (int i = 1; i <= n; i++) {  
            temp.add(i);  
        }  
        result.add(temp);  
        return result;  
    //if k==1  
    } else if (k == 1) {  
  
        for (int i = 1; i <= n; i++) {  
            ArrayList<Integer> temp = new ArrayList<Integer>();  
            temp.add(i);  
            result.add(temp);  
        }  
    }  
    return result;  
}
```

---

```
        result.add(temp);
    }

    return result;
}

//for normal cases, initialize a list with one element
for (int i = 1; i <= n - k + 1; i++) {
    ArrayList<Integer> temp = new ArrayList<Integer>();
    temp.add(i);
    result.add(temp);
}

//recursively add more elements
combine(n, k, result);

return result;
}

public void combine(int n, int k, ArrayList<ArrayList<Integer>> result) {
    ArrayList<ArrayList<Integer>> prevResult = new
        ArrayList<ArrayList<Integer>>();
    prevResult.addAll(result);

    if(result.get(0).size() == k) return;

    result.clear();
    for (ArrayList<Integer> one : prevResult) {

        for (int i = 1; i <= n; i++) {
            if (i > one.get(one.size() - 1)) {
                ArrayList<Integer> temp = new ArrayList<Integer>();
                temp.addAll(one);
                temp.add(i);
                result.add(temp);
            }
        }
    }

    combine(n, k, result);
}
```

---

### 229.3 Java Solution 2 - DFS

---

```
public ArrayList<ArrayList<Integer>> combine(int n, int k) {
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
```

```
if (n <= 0 || n < k)
    return result;

ArrayList<Integer> item = new ArrayList<Integer>();
dfs(n, k, 1, item, result); // because it need to begin from 1

return result;
}

private void dfs(int n, int k, int start, ArrayList<Integer> item,
    ArrayList<ArrayList<Integer>> res) {
    if (item.size() == k) {
        res.add(new ArrayList<Integer>(item));
        return;
    }

    for (int i = start; i <= n; i++) {
        item.add(i);
        dfs(n, k, i + 1, item, res);
        item.remove(item.size() - 1);
    }
}
```

---



# 230 Letter Combinations of a Phone Number

Given a digit string, return all possible letter combinations that the number could represent. (Check out your cellphone to see the mappings) Input:Digit string "23", Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].

## 230.1 Analysis

This problem can be solved by a typical DFS algorithm. DFS problems are very similar and can be solved by using a simple recursion. Check out [the index page](#) to see other DFS problems.

## 230.2 Java Solution

---

```
public List<String> letterCombinations(String digits) {
    HashMap<Integer, String> map = new HashMap<Integer, String>();
    map.put(2, "abc");
    map.put(3, "def");
    map.put(4, "ghi");
    map.put(5, "jkl");
    map.put(6, "mno");
    map.put(7, "pqrs");
    map.put(8, "tuv");
    map.put(9, "wxyz");
    map.put(0, "");

    ArrayList<String> result = new ArrayList<String>();

    if(digits == null || digits.length() == 0)
        return result;

    ArrayList<Character> temp = new ArrayList<Character>();
    getString(digits, temp, result, map);

    return result;
}

public void getString(String digits, ArrayList<Character> temp,
    ArrayList<String> result, HashMap<Integer, String> map){
```

## 230 Letter Combinations of a Phone Number

---

```
if(digits.length() == 0){
    char[] arr = new char[temp.size()];
    for(int i=0; i<temp.size(); i++){
        arr[i] = temp.get(i);
    }
    result.add(String.valueOf(arr));
    return;
}

Integer curr = Integer.valueOf(digits.substring(0,1));
String letters = map.get(curr);
for(int i=0; i<letters.length(); i++){
    temp.add(letters.charAt(i));
    getString(digits.substring(1), temp, result, map);
    temp.remove(temp.size()-1);
}
}
```

---

## 231 Restore IP Addresses

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example: given "25525511135", return ["255.255.11.135", "255.255.111.35"].

### 231.1 Java Solution

This is a typical search problem and it can be solved by using DFS.

```
public List<String> restoreIpAddresses(String s) {
    ArrayList<ArrayList<String>> result = new ArrayList<ArrayList<String>>();
    ArrayList<String> t = new ArrayList<String>();
    dfs(result, s, 0, t);

    ArrayList<String> finalResult = new ArrayList<String>();

    for(ArrayList<String> l: result){
        StringBuilder sb = new StringBuilder();
        for(String str: l){
            sb.append(str+".");
        }
        sb.setLength(sb.length() - 1);
        finalResult.add(sb.toString());
    }

    return finalResult;
}

public void dfs(ArrayList<ArrayList<String>> result, String s, int start,
    ArrayList<String> t){
    //if already get 4 numbers, but s is not consumed, return
    if(t.size()>=4 && start!=s.length())
        return;

    //make sure t's size + remaining string's length >=4
    if((t.size()+s.length()-start+1)<4)
        return;

    //t's size is 4 and no remaining part that is not consumed.
    if(t.size()==4 && start==s.length()){
        ArrayList<String> temp = new ArrayList<String>(t);
        result.add(temp);
    }
}
```

## 231 Restore IP Addresses

---

```
        return;
    }

    for(int i=1; i<=3; i++){
        //make sure the index is within the boundary
        if(start+i>s.length())
            break;

        String sub = s.substring(start, start+i);
        //handle case like 001. i.e., if length > 1 and first char is 0, ignore
        //the case.
        if(i>1 && s.charAt(start)=='0'){
            break;
        }

        //make sure each number <= 255
        if(Integer.valueOf(sub)>255)
            break;

        t.add(sub);
        dfs(result, s, start+i, t);
        t.remove(t.size()-1);
    }
}
```

---

# 232 Reverse Integer

LeetCode - Reverse Integer:

*Reverse digits of an integer. Example1:  $x = 123$ , return  $321$  Example2:  $x = -123$ , return  $-321$*

## 232.1 Naive Method

We can convert the integer to a string/char array, reverse the order, and convert the string/char array back to an integer. However, this will require extra space for the string. It doesn't seem to be the right way, if you come with such a solution.

## 232.2 Efficient Approach

Actually, this can be done by using the following code.

---

```
public int reverse(int x) {
    //flag marks if x is negative
    boolean flag = false;
    if (x < 0) {
        x = 0 - x;
        flag = true;
    }

    int res = 0;
    int p = x;

    while (p > 0) {
        int mod = p % 10;
        p = p / 10;
        res = res * 10 + mod;
    }

    if (flag) {
        res = 0 - res;
    }

    return res;
}
```

---

### 232.3 Succinct Solution

This solution is from Sherry, it is succinct and it is pretty.

---

```
public int reverse(int x) {
    int rev = 0;
    while(x != 0){
        rev = rev*10 + x%10;
        x = x/10;
    }

    return rev;
}
```

---

### 232.4 Handle Out of Range Problem

As we form a new integer, it is possible that the number is out of range. We can use the following code to assign the newly formed integer. When it is out of range, throw an exception.

---

```
try{
    result = ...;
}catch(InputMismatchException exception){
    System.out.println("This is not an integer");
}
```

---

Please leave your comment if there is any better solutions.

# 233 Palindrome Number

Determine whether an integer is a palindrome. Do this without extra space.

## 233.1 Thoughts

Problems related with numbers are frequently solved by / and %

Note: no extra space here means do not convert the integer to string, since string will be a copy of the integer and take extra space. The space taken by div, left, and right can be ignored.

## 233.2 Java Solution

---

```
public class Solution {
    public boolean isPalindrome(int x) {
        //negative numbers are not palindrome
        if (x < 0)
            return false;

        // initialize how many zeros
        int div = 1;
        while (x / div >= 10) {
            div *= 10;
        }

        while (x != 0) {
            int left = x / div;
            int right = x % 10;

            if (left != right)
                return false;

            x = (x % div) / 10;
            div /= 100;
        }

        return true;
    }
}
```

---



## 234 Pow(x, n)

Problem:

*Implement pow(x, n).*

This is a great example to illustrate how to solve a problem during a technical interview. The first and second solution exceeds time limit; the third and fourth are accepted.

### 234.1 Naive Method

First of all, assuming n is not negative, to calculate x to the power of n, we can simply multiply x n times, i.e.,  $x * x * \dots * x$ . The time complexity is  $O(n)$ . The implementation is as simple as:

---

```
public class Solution {
    public double pow(double x, int n) {
        if(x == 0) return 0;
        if(n == 0) return 1;

        double result=1;
        for(int i=1; i<=n; i++){
            result = result * x;
        }

        return result;
    }
}
```

---

Now we should think about how to do better than  $O(n)$ .

### 234.2 Recursive Method

Naturally, we next may think how to do it in  $O(\log n)$ . We have a relation that  $x^n = x^{(n/2)} * x^{(n/2)} * x^{(n)}$

---

```
public static double pow(double x, int n) {
    if(n == 0)
        return 1;

    if(n == 1)
        return x;
```

---

```

int half = n/2;
int remainder = n%2;

if(n % 2 ==1 && x < 0 && n < 0)
    return - 1/(pow(-x, half) * pow(-x, half) * pow(-x, remainder));
else if (n < 0)
    return 1/(pow(x, -half) * pow(x, -half) * pow(x, -remainder));
else
    return (pow(x, half) * pow(x, half) * pow(x, remainder));
}

```

---

### 234.3 In this solution, we can handle cases that x <0 and n <0. This solution actually takes more time than the first solution. Why? 3. Accepted Solution

The accepted solution is also recursive, but does division first. Time complexity is O(nlog(n)). The key part of solving this problem is the while loop.

---

```

public double pow(double x, int n) {
    if (n == 0)
        return 1;
    if (n == 1)
        return x;

    int pn = n > 0 ? n : -n;// positive n
    int pn2 = pn;

    double px = x > 0 ? x : -x;// positive x
    double result = px;

    int k = 1;
    //the key part of solving this problem
    while (pn / 2 > 0) {
        result = result * result;
        pn = pn / 2;
        k = k * 2;
    }

    result = result * pow(px, pn2 - k);

    // handle negative result
    if (x < 0 && n % 2 == 1)
        result = -result;

    // handle negative power
    if (n < 0)

```

---

```
    result = 1 / result;

    return result;
}
```

---

## 234.4 Best Solution

The most understandable solution I have found so far.

---

```
public double power(double x, int n) {
    if (n == 0)
        return 1;

    double v = power(x, n / 2);

    if (n % 2 == 0) {
        return v * v;
    } else {
        return v * v * x;
    }
}

public double pow(double x, int n) {
    if (n < 0) {
        return 1 / power(x, -n);
    } else {
        return power(x, n);
    }
}
```

---