

# Coding Interview in Java

Program Creek

# Contents

1	Remove Duplicates from Sorted Array	13
2	Remove Duplicates from Sorted Array II	15
3	Remove Element	17
4	Move Zeroes	18
5	Candy	19
6	Trapping Rain Water	21
7	Product of Array Except Self	23
8	Minimum Size Subarray Sum	25
9	Summary Ranges	27
10	Merge Intervals	28
11	Insert Interval	29
12	Partition Labels	32
13	One Edit Distance	34
14	Merge Sorted Array	35
15	Is Subsequence	37
16	Container With Most Water	38
17	Reverse Vowels of a String	39
18	Shortest Word Distance	40
19	Shortest Word Distance II	41
20	Shortest Word Distance III	43
21	Intersection of Two Arrays	45
22	Intersection of Two Arrays II	47
23	Two Sum II Input array is sorted	49
24	Two Sum III Data structure design	50

---

<b>25</b>	<b>3Sum</b>	<b>51</b>
<b>26</b>	<b>4Sum</b>	<b>53</b>
<b>27</b>	<b>3Sum Closest</b>	<b>55</b>
<b>28</b>	<b>Search Insert Position</b>	<b>56</b>
<b>29</b>	<b>Median of Two Sorted Arrays</b>	<b>58</b>
<b>30</b>	<b>Find Minimum in Rotated Sorted Array</b>	<b>60</b>
<b>31</b>	<b>Find Minimum in Rotated Sorted Array II</b>	<b>62</b>
<b>32</b>	<b>Find First and Last Position of Element in Sorted Array</b>	<b>64</b>
<b>33</b>	<b>Guess Number Higher or Lower</b>	<b>66</b>
<b>34</b>	<b>First Bad Version</b>	<b>68</b>
<b>35</b>	<b>Search in Rotated Sorted Array</b>	<b>70</b>
<b>36</b>	<b>Search in Rotated Sorted Array II</b>	<b>72</b>
<b>37</b>	<b>Longest Increasing Subsequence</b>	<b>73</b>
<b>38</b>	<b>Count of Smaller Numbers After Self</b>	<b>76</b>
<b>39</b>	<b>Russian Doll Envelopes</b>	<b>79</b>
<b>40</b>	<b>HIndex</b>	<b>81</b>
<b>41</b>	<b>HIndex II</b>	<b>83</b>
<b>42</b>	<b>Valid Anagram</b>	<b>84</b>
<b>43</b>	<b>Group Shifted Strings</b>	<b>86</b>
<b>44</b>	<b>Palindrome Pairs</b>	<b>88</b>
<b>45</b>	<b>Line Reflection</b>	<b>90</b>
<b>46</b>	<b>Isomorphic Strings</b>	<b>91</b>
<b>47</b>	<b>Two Sum</b>	<b>92</b>
<b>48</b>	<b>Maximum Size Subarray Sum Equals k</b>	<b>93</b>
<b>49</b>	<b>Subarray Sum Equals K</b>	<b>95</b>
<b>50</b>	<b>Maximum Subarray</b>	<b>96</b>
<b>51</b>	<b>Maximum Product Subarray</b>	<b>99</b>
<b>52</b>	<b>Longest Substring Without Repeating Characters</b>	<b>100</b>
<b>53</b>	<b>Longest Substring with At Most K Distinct Characters</b>	<b>102</b>
<b>54</b>	<b>Substring with Concatenation of All Words</b>	<b>104</b>

---

<b>55 Minimum Window Substring</b>	<b>106</b>
<b>56 Longest Substring with At Least K Repeating Characters</b>	<b>108</b>
<b>57 Longest Consecutive Sequence</b>	<b>110</b>
<b>58 Majority Element</b>	<b>112</b>
<b>59 Majority Element II</b>	<b>114</b>
<b>60 Increasing Triplet Subsequence</b>	<b>115</b>
<b>61 Find the Second Largest Number in an Array</b>	<b>117</b>
<b>62 Word Ladder</b>	<b>118</b>
<b>63 Word Ladder II</b>	<b>120</b>
<b>64 Top K Frequent Elements</b>	<b>122</b>
<b>65 Meeting Rooms II</b>	<b>125</b>
<b>66 Meeting Rooms</b>	<b>127</b>
<b>67 Range Addition</b>	<b>128</b>
<b>68 Merge K Sorted Arrays in Java</b>	<b>130</b>
<b>69 Merge k Sorted Lists</b>	<b>132</b>
<b>70 Rearrange String k Distance Apart</b>	<b>133</b>
<b>71 Contains Duplicate</b>	<b>135</b>
<b>72 Contains Duplicate II</b>	<b>136</b>
<b>73 Contains Duplicate III</b>	<b>137</b>
<b>74 Max Sum of Rectangle No Larger Than K</b>	<b>139</b>
<b>75 Maximum Sum of Subarray Close to K</b>	<b>142</b>
<b>76 Sliding Window Maximum</b>	<b>144</b>
<b>77 Moving Average from Data Stream</b>	<b>146</b>
<b>78 Find Median from Data Stream</b>	<b>147</b>
<b>79 Data Stream as Disjoint Intervals</b>	<b>149</b>
<b>80 Linked List Random Node</b>	<b>151</b>
<b>81 Shuffle an Array</b>	<b>152</b>
<b>82 Sort List</b>	<b>154</b>
<b>83 Quicksort Array in Java</b>	<b>156</b>
<b>84 Kth Largest Element in an Array</b>	<b>159</b>

---

<b>85 Sort Colors</b>	<b>161</b>
<b>86 Maximum Gap</b>	<b>162</b>
<b>87 Group Anagrams</b>	<b>164</b>
<b>88 Clone Graph</b>	<b>165</b>
<b>89 Course Schedule</b>	<b>168</b>
<b>90 Course Schedule II</b>	<b>171</b>
<b>91 Minimum Height Trees</b>	<b>173</b>
<b>92 Reconstruct Itinerary</b>	<b>175</b>
<b>93 Graph Valid Tree</b>	<b>176</b>
<b>94 Ugly Number</b>	<b>178</b>
<b>95 Ugly Number II</b>	<b>179</b>
<b>96 Super Ugly Number</b>	<b>180</b>
<b>97 Find K Pairs with Smallest Sums</b>	<b>181</b>
<b>98 Rotate Array in Java</b>	<b>182</b>
<b>99 Reverse Words in a String II</b>	<b>185</b>
<b>100 Missing Number</b>	<b>186</b>
<b>101 Find the Duplicate Number</b>	<b>187</b>
<b>102 First Missing Positive</b>	<b>189</b>
<b>103 Queue Reconstruction by Height</b>	<b>191</b>
<b>104 Binary Watch</b>	<b>192</b>
<b>105 Search a 2D Matrix</b>	<b>195</b>
<b>106 Search a 2D Matrix II</b>	<b>196</b>
<b>107 Kth Smallest Element in a Sorted Matrix</b>	<b>198</b>
<b>108 Design Snake Game</b>	<b>200</b>
<b>109 Number of Islands II</b>	<b>202</b>
<b>110 Number of Connected Components in an Undirected Graph</b>	<b>204</b>
<b>111 Longest Increasing Path in a Matrix</b>	<b>207</b>
<b>112 Word Search</b>	<b>210</b>
<b>113 Word Search II</b>	<b>213</b>
<b>114 Number of Islands</b>	<b>217</b>

---

<b>115 Find a Path in a Matrix</b>	220
<b>116 Sudoku Solver</b>	222
<b>117 Valid Sudoku</b>	225
<b>118 Walls and Gates</b>	227
<b>119 Surrounded Regions</b>	230
<b>120 Set Matrix Zeros</b>	234
<b>121 Spiral Matrix</b>	237
<b>122 Spiral Matrix II</b>	241
<b>123 Rotate Image</b>	243
<b>124 Range Sum Query 2D Immutable</b>	244
<b>125 Shortest Distance from All Buildings</b>	247
<b>126 Best Meeting Point</b>	249
<b>127 Game of Life</b>	250
<b>128 TicTacToe</b>	252
<b>129 Sparse Matrix Multiplication</b>	255
<b>130 Add Two Numbers</b>	257
<b>131 Reorder List</b>	259
<b>132 Linked List Cycle</b>	263
<b>133 Copy List with Random Pointer</b>	264
<b>134 Merge Two Sorted Lists</b>	266
<b>135 Odd Even Linked List</b>	268
<b>136 Remove Duplicates from Sorted List</b>	270
<b>137 Remove Duplicates from Sorted List II</b>	272
<b>138 Partition List</b>	273
<b>139 Intersection of Two Linked Lists</b>	274
<b>140 Remove Linked List Elements</b>	276
<b>141 Swap Nodes in Pairs</b>	277
<b>142 Reverse Linked List</b>	279
<b>143 Reverse Linked List II</b>	281
<b>144 Reverse Double Linked List</b>	283

---

<b>145 Remove Nth Node From End of List</b>	285
<b>146 Palindrome Linked List</b>	287
<b>147 Delete Node in a Linked List</b>	290
<b>148 Reverse Nodes in kGroup</b>	291
<b>149 Plus One Linked List</b>	293
<b>150 Binary Tree Preorder Traversal</b>	295
<b>151 Binary Tree Inorder Traversal</b>	296
<b>152 Binary Tree Postorder Traversal</b>	298
<b>153 Binary Tree Level Order Traversal</b>	301
<b>154 Binary Tree Level Order Traversal II</b>	303
<b>155 Binary Tree Vertical Order Traversal</b>	305
<b>156 Invert Binary Tree</b>	307
<b>157 Kth Smallest Element in a BST</b>	308
<b>158 Binary Tree Longest Consecutive Sequence</b>	310
<b>159 Validate Binary Search Tree</b>	312
<b>160 Flatten Binary Tree to Linked List</b>	314
<b>161 Path Sum</b>	316
<b>162 Path Sum II</b>	318
<b>163 Construct Binary Tree from Inorder and Postorder Traversal</b>	320
<b>164 Construct Binary Tree from Preorder and Inorder Traversal</b>	322
<b>165 Convert Sorted Array to Binary Search Tree</b>	324
<b>166 Convert Sorted List to Binary Search Tree</b>	325
<b>167 Minimum Depth of Binary Tree</b>	327
<b>168 Binary Tree Maximum Path Sum</b>	329
<b>169 Balanced Binary Tree</b>	330
<b>170 Symmetric Tree</b>	332
<b>171 Binary Search Tree Iterator</b>	333
<b>172 Binary Tree Right Side View</b>	335
<b>173 Lowest Common Ancestor of a Binary Search Tree</b>	337
<b>174 Lowest Common Ancestor of a Binary Tree</b>	338

---

<b>175 Most Frequent Subtree Sum</b>	340
<b>176 Verify Preorder Serialization of a Binary Tree</b>	342
<b>177 Populating Next Right Pointers in Each Node</b>	344
<b>178 Populating Next Right Pointers in Each Node II</b>	347
<b>179 Unique Binary Search Trees</b>	349
<b>180 Unique Binary Search Trees II</b>	351
<b>181 Sum Root to Leaf Numbers</b>	353
<b>182 Count Complete Tree Nodes</b>	355
<b>183 Closest Binary Search Tree Value</b>	357
<b>184 Binary Tree Paths</b>	359
<b>185 Maximum Depth of Binary Tree</b>	361
<b>186 Recover Binary Search Tree</b>	362
<b>187 Same Tree</b>	363
<b>188 Serialize and Deserialize Binary Tree</b>	364
<b>189 Inorder Successor in BST</b>	367
<b>190 Inorder Successor in BST II</b>	369
<b>191 Find Leaves of Binary Tree</b>	371
<b>192 Largest BST Subtree</b>	373
<b>193 Implement Trie (Prefix Tree)</b>	375
<b>194 Add and Search Word Data structure design</b>	379
<b>195 Range Sum Query Mutable</b>	383
<b>196 The Skyline Problem</b>	387
<b>197 Implement Stack using Queues</b>	389
<b>198 Implement Queue using Stacks</b>	391
<b>199 Implement a Stack Using an Array in Java</b>	392
<b>200 Implement a Queue using an Array in Java</b>	394
<b>201 Evaluate Reverse Polish Notation</b>	396
<b>202 Valid Parentheses</b>	399
<b>203 Longest Valid Parentheses</b>	400
<b>204 Valid Palindrome</b>	401

---

<b>205 Min Stack</b>	404
<b>206 Max Chunks To Make Sorted</b>	406
<b>207 Largest Rectangle in Histogram</b>	407
<b>208 Maximal Rectangle</b>	409
<b>209 Mini Parser</b>	411
<b>210 Flatten Nested List Iterator</b>	413
<b>211 Nested List Weight Sum</b>	415
<b>212 Longest Absolute File Path</b>	417
<b>213 Decode String</b>	419
<b>214 Partition to K Equal Sum Subsets</b>	421
<b>215 Permutations</b>	423
<b>216 Permutations II</b>	426
<b>217 Permutation Sequence</b>	428
<b>218 Number of Squareful Arrays</b>	430
<b>219 Generate Parentheses</b>	433
<b>220 Combination Sum</b>	435
<b>221 Combination Sum II</b>	437
<b>222 Combination Sum III</b>	438
<b>223 Combination Sum IV</b>	439
<b>224 Wildcard Matching</b>	440
<b>225 Regular Expression Matching in Java</b>	441
<b>226 Get Target Number Using Number List and Arithmetic Operations</b>	444
<b>227 Flip Game</b>	446
<b>228 Flip Game II</b>	447
<b>229 Word Pattern</b>	448
<b>230 Word Pattern II</b>	449
<b>231 Scramble String</b>	451
<b>232 Remove Invalid Parentheses</b>	452
<b>233 Shortest Palindrome</b>	454
<b>234 Lexicographical Numbers</b>	456

---

<b>235 Combinations</b>	458
<b>236 Letter Combinations of a Phone Number</b>	459
<b>237 Restore IP Addresses</b>	461
<b>238 Factor Combinations</b>	463
<b>239 Subsets</b>	464
<b>240 Subsets II</b>	466
<b>241 Coin Change</b>	468
<b>242 Palindrome Partitioning</b>	471
<b>243 Palindrome Partitioning II</b>	473
<b>244 House Robber</b>	474
<b>245 House Robber II</b>	476
<b>246 House Robber III</b>	477
<b>247 Jump Game</b>	478
<b>248 Jump Game II</b>	480
<b>249 Best Time to Buy and Sell Stock</b>	481
<b>250 Best Time to Buy and Sell Stock II</b>	482
<b>251 Best Time to Buy and Sell Stock III</b>	483
<b>252 Best Time to Buy and Sell Stock IV</b>	485
<b>253 Dungeon Game</b>	487
<b>254 Decode Ways</b>	488
<b>255 Perfect Squares</b>	489
<b>256 Word Break</b>	490
<b>257 Word Break II</b>	493
<b>258 Minimum Window Subsequence</b>	496
<b>259 Maximal Square</b>	497
<b>260 Minimum Path Sum</b>	499
<b>261 Unique Paths</b>	501
<b>262 Unique Paths II</b>	503
<b>263 Paint House</b>	505
<b>264 Paint House II</b>	507

---

<b>265 Edit Distance in Java</b>	509
<b>266 Distinct Subsequences Total</b>	512
<b>267 Longest Palindromic Substring</b>	514
<b>268 Longest Common Subsequence</b>	516
<b>269 Longest Common Substring</b>	518
<b>270 LRU Cache</b>	520
<b>271 Insert Delete GetRandom O(1)</b>	523
<b>272 Insert Delete GetRandom O(1) Duplicates allowed</b>	525
<b>273 Design a Data Structure with Insert, Delete and GetMostFrequent of O(1)</b>	527
<b>274 Design Phone Directory</b>	530
<b>275 Design Twitter</b>	531
<b>276 Single Number</b>	534
<b>277 Single Number II</b>	535
<b>278 Twitter Codility Problem Max Binary Gap</b>	536
<b>279 Number of 1 Bits</b>	538
<b>280 Reverse Bits</b>	539
<b>281 Repeated DNA Sequences</b>	540
<b>282 Bitwise AND of Numbers Range</b>	542
<b>283 Sum of Two Integers</b>	543
<b>284 Counting Bits</b>	544
<b>285 Maximum Product of Word Lengths</b>	546
<b>286 Gray Code</b>	547
<b>287 UTF8 Validation</b>	548
<b>288 Pow(x, n)</b>	549

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/10-algorithms-for-coding-interview/>

# 1 Remove Duplicates from Sorted Array

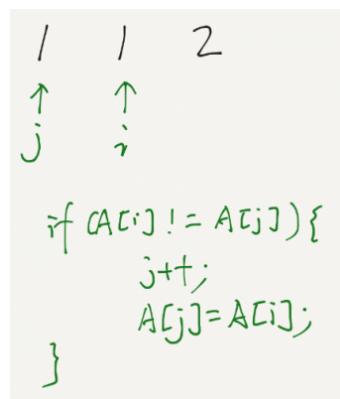
Given a sorted array, remove the duplicates in place such that each element appear only once and return the new length. Do not allocate extra space for another array, you must do this in place with constant memory.

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

## 1.1 Analysis

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

## 1.2 Java Solution



---

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

    int j = 0;
    int i = 1;

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

        i++;
    }

    return j + 1;
}
```

---

Note that we only care about the first unique part of the original array. So it is ok if input array is 1, 2, 2, 3, 3, the array is changed to 1, 2, 3, 3, 3.

## 2 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.

### 2.1 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 int removeDuplicates(int[] nums) {
    if(nums==null){
        return 0;
    }
    if(nums.length<3){
        return nums.length;
    }

    int i=0;
    int j=1;
    /*
        i, j  1 1 1 2 2 3
    step1 0 1      i j
    step2 1 2      i j
    step3 1 3      i j
    step4 2 4      i j
    */

    while(j<nums.length){
        if(nums[j]==nums[i]){
            if(i==0){
                i++;
                j++;
            }else if(nums[i]==nums[i-1]){
                j++;
            }else{
                i++;
                nums[i]=nums[j];
                j++;
            }
        }else{
            i++;
            nums[i]=nums[j];
            j++;
        }
    }

    return i+1;
}
```

---

The problem with this solution is that there are 4 cases to handle. If we shift our two points to right by 1 element, the solution can be simplified as the Solution 2.

## 2.2 Java Solution 2

---

```
public int removeDuplicates(int[] nums) {
    if(nums==null){
        return 0;
    }

    if (nums.length <= 2){
        return nums.length;
    }
/*
1,1,1,2,2,3
 i j
*/
    int i = 1; // point to previous
    int j = 2; // point to current

    while (j < nums.length) {
        if (nums[j] == nums[i] && nums[j] == nums[i - 1]) {
            j++;
        } else {
            i++;
            nums[i] = nums[j];
            j++;
        }
    }
    return i + 1;
}
```

---

# 3 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.)

## 3.1 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;
}
```

---

## 4 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]`.

### 4.1 Java Solution 2

We can use the similar code that is used to solve [Remove Duplicates from Sorted Array I, II](#), [Remove Element](#).

---

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

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

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

---

# 5 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?

## 5.1 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](#).

## 5.2 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];

    //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;  
}
```

---

# 6 Trapping Rain Water

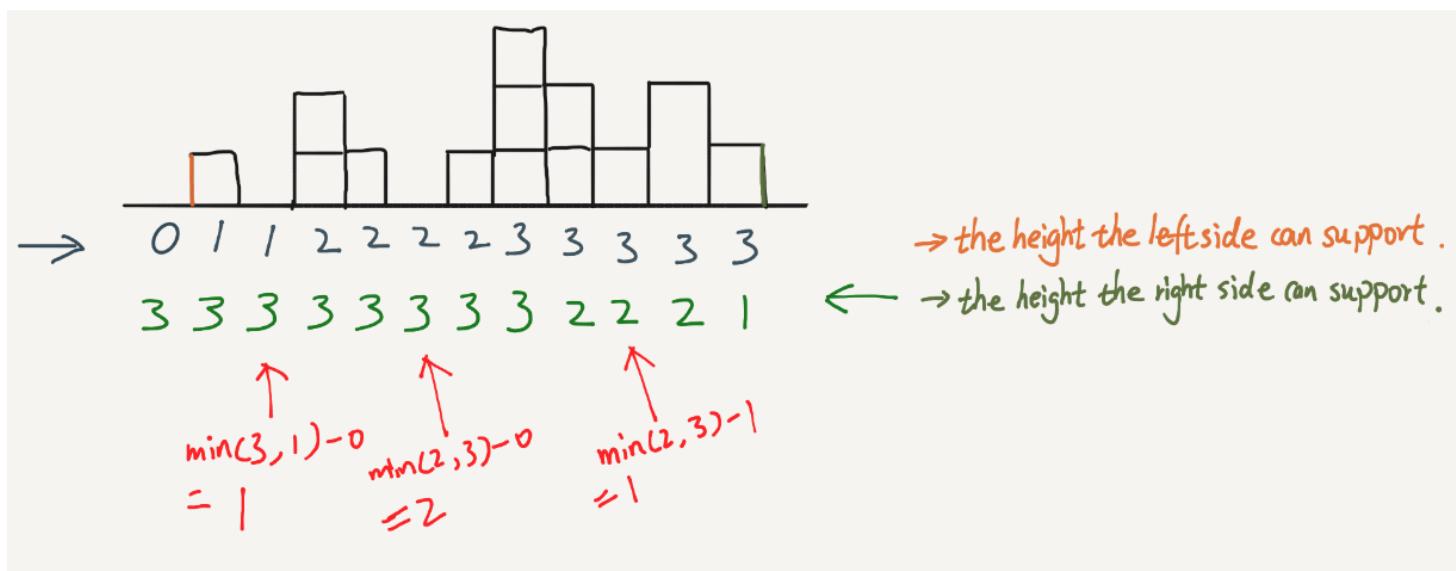
Given  $n$  non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

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

## 6.1 Analysis

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

## 6.2 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];
        }
    }

    //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 {
            right[i]=height[i];
        }
    }

    for(int i=1; i<height.length-1; i++){
        result += Math.min(left[i], right[i]) - height[i];
    }
}
```

```
        }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{
        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;
}
```

---

# 7 Product of Array Except Self

Given an array of n integers where  $n > 1$ , `nums`, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

Solve it without division and in  $O(n)$ .

For example, given `[1,2,3,4]`, return `[24,12,8,6]`.

## 7.1 Java Solution 1

---

```
public int[] productExceptSelf(int[] nums) {
    int[] result = new int[nums.length];

    int[] t1 = new int[nums.length];
    int[] t2 = new int[nums.length];

    t1[0]=1;
    t2[nums.length-1]=1;

    //scan from left to right
    for(int i=0; i<nums.length-1; i++){
        t1[i+1] = nums[i] * t1[i];
    }

    //scan from right to left
    for(int i=nums.length-1; i>0; i--){
        t2[i-1] = t2[i] * nums[i];
    }

    //multiply
    for(int i=0; i<nums.length; i++){
        result[i] = t1[i] * t2[i];
    }

    return result;
}
```

---

## 7.2 Java Solution 2

We can directly put the product values into the final result array. This saves the extra space to store the 2 intermediate arrays in Solution 1.

---

```
public int[] productExceptSelf(int[] nums) {
    int[] result = new int[nums.length];

    result[nums.length-1]=1;
    for(int i=nums.length-2; i>=0; i--){
        result[i]=result[i+1]*nums[i+1];
    }
```

---

```
int left=1;
for(int i=0; i<nums.length; i++){
    result[i]=result[i]*left;
    left = left*nums[i];
}

return result;
}
```

---

# 8 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.

## 8.1 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.

## 8.2 Java Solution - two pointers

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;
```

---

}

Similarly, we can also write it in a more readable way.

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

    int i=0;
    int j=0;
    int sum=0;

    int minLen = Integer.MAX_VALUE;

    while(j<nums.length){
        if(sum<s){
            sum += nums[j];
            j++;
        }else{
            minLen = Math.min(minLen, j-i);
            if(i==j-1)
                return 1;

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

    while(sum>=s){
        minLen = Math.min(minLen, j-i);

        sum -=nums[i++];
    }

    return minLen==Integer.MAX_VALUE? 0: minLen;
}
```

---

# 9 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"].

## 9.1 Analysis

When iterating over the array, two values need to be tracked: 1) the first value of a new range and 2) the previous value in the range.

## 9.2 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];
        }

        pre = nums[i];
    }

    return result;
}
```

---

# 10 Merge Intervals

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].

## 10.1 Analysis

The key to solve this problem is defining a Comparator first to sort the arraylist of Intevals.

## 10.2 Java Solution

---

```
public List<Interval> merge(List<Interval> intervals) {
    List<Interval> result = new ArrayList<>();
    if(intervals==null || intervals.size()==0){
        return result;
    }

    Comparator<Interval> comp = Comparator.comparing((Interval i)->i.start);
    Collections.sort(intervals, comp);

    Interval temp = intervals.get(0);
    for(int i=1; i<intervals.size(); i++){
        Interval curr = intervals.get(i);
        if (temp.end>=curr.start){
            temp.end = Math.max(curr.end, temp.end);
        }else{
            result.add(temp);
            temp = curr;
        }
    }

    result.add(temp);

    return result;
}
```

---

# 11 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].

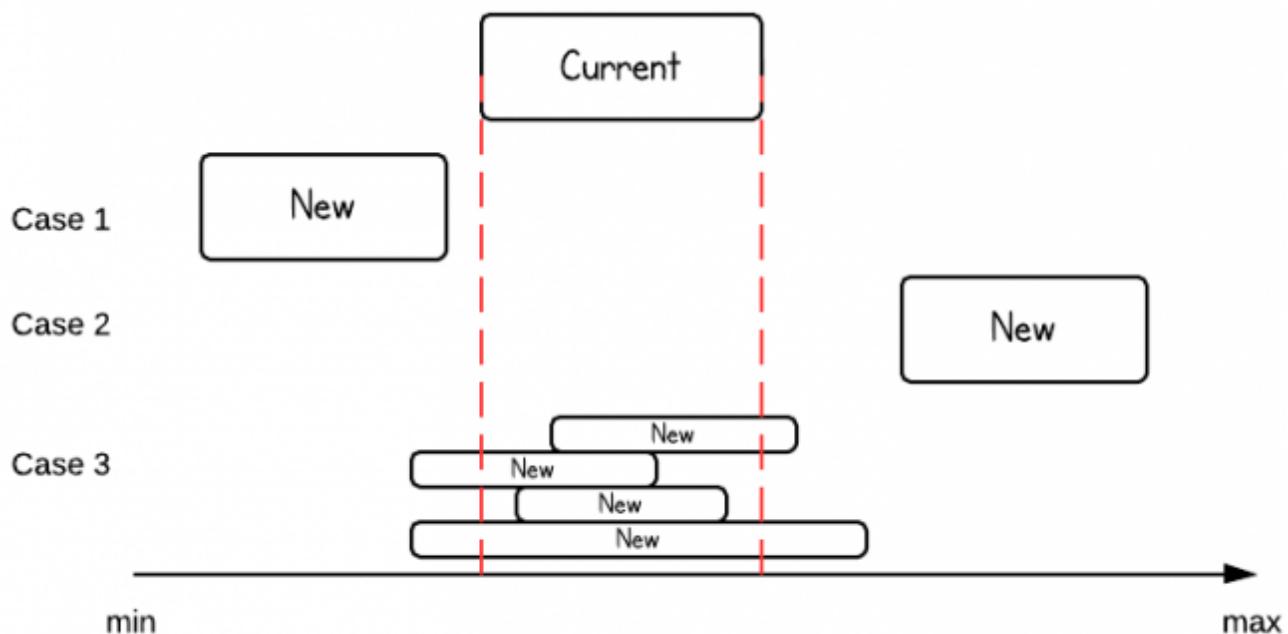
Example 2:

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].

## 11.1 Java Solution 1

When iterating over the list, there are three cases for the current range.



```
/**  
 * Definition for an interval.  
 * public class Interval {  
 *     int start;  
 *     int end;  
 *     Interval() { start = 0; end = 0; }  
 *     Interval(int s, int e) { start = s; end = e; }
```

```

* }
*/
public class Solution {
    public ArrayList<Interval> insert(ArrayList<Interval> intervals, Interval newInterval) {

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

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

        result.add(newInterval);

        return result;
    }
}

```

---

## 11.2 Java Solution 2 - Binary Search

If the intervals list is an ArrayList, we can use binary search to make the best search time complexity O(log(n)). However, the worst time is bounded by shifting the array list if a new range needs to be inserted. So time complexity is still O(n).

```

public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
    List<Interval> result = new ArrayList<>();

    if (intervals.size() == 0) {
        result.add(newInterval);
        return result;
    }

    int p = helper(intervals, newInterval);
    result.addAll(intervals.subList(0, p));

    for (int i = p; i < intervals.size(); i++) {
        Interval interval = intervals.get(i);
        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;
}

public int helper(List<Interval> intervals, Interval newInterval) {
    int low = 0;
    int high = intervals.size() - 1;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if (newInterval.start <= intervals.get(mid).start) {
            high = mid;
        } else {
            low = mid + 1;
        }
    }

    return high == 0 ? 0 : high - 1;
}
```

---

The best time is  $O(\log(n))$  and worst case time is  $O(n)$ .

## 12 Partition Labels

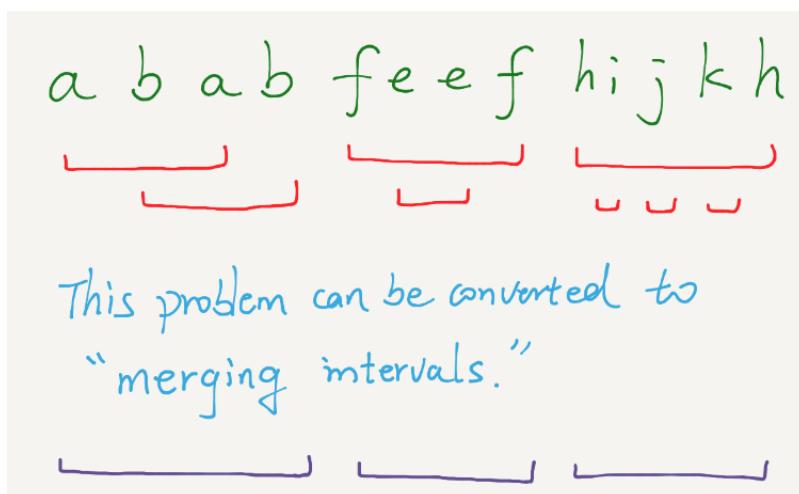
A string S of lowercase letters is given. We want to partition this string into as many parts as possible so that each letter appears in at most one part, and return a list of integers representing the size of these parts.

For example:

Input: S = "ababfeefhijkh" Output: [4,4,5]

Explanation: The partition is "abab", "feef", "hijkh". This is a partition so that each letter appears in at most one part.

### 12.1 Java Solution



```
public List<Integer> partitionLabels(String S) {  
    ArrayList<Integer> result = new ArrayList<>();  
  
    HashMap<Character, int[]> map = new HashMap<>();  
    for(int i=0; i<S.length(); i++){  
        char c = S.charAt(i);  
        int[] arr = map.get(c);  
  
        if(arr == null){  
            arr = new int[]{i, i};  
            map.put(c, arr);  
        }else{  
            arr[1]=i;  
        }  
    }  
  
    ArrayList<int[]> list = new ArrayList<>();  
    list.addAll(map.values());  
  
    Collections.sort(list, Comparator.comparing((int[] arr) -> arr[0]));
```

```
int[] t = list.get(0);
for(int i=1; i<list.size(); i++){
    int[] range = list.get(i);

    if(range[1]<=t[1]){
        continue;
    }else if(range[0]>t[1]){ //impossible be equal
        result.add(t[1]-t[0]+1);
        t = range;
    }else{
        t[1] = range[1];
    }
}

result.add(t[1]-t[0]+1);

return result;
}
```

---

# 13 One Edit Distance

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

## 13.1 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++;
    }

    if(count==1)
        return true;

    return false;
}
```

---

# 14 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.

## 14.1 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.

## 14.2 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--;
        }
    }
}
```

---

## 14.3 Java Solution 2

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

---

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

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

```
    else
        A[k--] = B[j--];
}
}
```

---

# 15 Is Subsequence

Given a string s and a string t, check if s is subsequence of t.

You may assume that there is only lower case English letters in both s and t. t is potentially a very long (length = 500,000) string, and s is a short string (<=100).

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).

## 15.1 Java Solution

---

```
public boolean isSubsequence(String s, String t) {  
    if(s.length()==0)  
        return true;  
  
    int i=0;  
    int j=0;  
    while(i<s.length() && j<t.length()){  
        if(s.charAt(i)==t.charAt(j)){  
            i++;  
        }  
  
        j++;  
  
        if(i==s.length())  
            return true;  
    }  
  
    return false;  
}
```

---

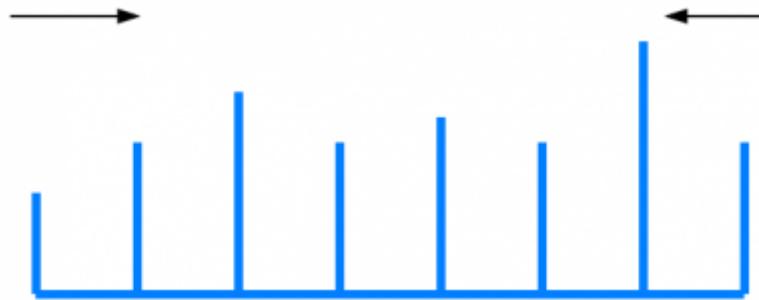
# 16 Container With Most Water

## 16.1 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.

## 16.2 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.



## 16.3 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]));
        if (height[left] < height[right])
            left++;
        else
            right--;
    }

    return max;
}
```

# 17 Reverse Vowels of a String

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

## 17.1 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--;
    }

    return new String(arr);
}
```

---

# 18 Shortest Word Distance

Given a list of words and two words word<sub>1</sub> and word<sub>2</sub>, return the shortest distance between these two words in the list.

For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word<sub>1</sub> = "coding", word<sub>2</sub> = "practice", return 3. Given word<sub>1</sub> = "makes", word<sub>2</sub> = "coding", return 1.

## 18.1 Java Solution

---

```
public int shortestDistance(String[] words, String word1, String word2) {
    int m=-1;
    int n=-1;

    int min = Integer.MAX_VALUE;

    for(int i=0; i<words.length; i++){
        String s = words[i];
        if(word1.equals(s)){
            m = i;
            if(n!=-1)
                min = Math.min(min, m-n);
        }else if(word2.equals(s)){
            n = i;
            if(m!=-1)
                min = Math.min(min, n-m);
        }
    }

    return min;
}
```

---

# 19 Shortest Word Distance II

This is a follow up of Shortest Word Distance. The only difference is now you are given the list of words and your method will be called repeatedly many times with different parameters. How would you optimize it?

Design a class which receives a list of words in the constructor, and implements a method that takes two words word<sub>1</sub> and word<sub>2</sub> and return the shortest distance between these two words in the list.

For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word<sub>1</sub> = "coding", word<sub>2</sub> = "practice", return 3. Given word<sub>1</sub> = "makes", word<sub>2</sub> = "coding", return 1.

## 19.1 Java Solution

---

```
public class WordDistance {
    HashMap<String, ArrayList<Integer>> map;
    public WordDistance(String[] words) {
        map = new HashMap<String, ArrayList<Integer>>();
        for(int i=0; i<words.length; i++){
            if(map.containsKey(words[i])){
                map.get(words[i]).add(i);
            }else{
                ArrayList<Integer> list = new ArrayList<Integer>();
                list.add(i);
                map.put(words[i], list);
            }
        }
    }

    public int shortest(String word1, String word2) {

        ArrayList<Integer> l1 = map.get(word1);
        ArrayList<Integer> l2 = map.get(word2);

        int result = Integer.MAX_VALUE;
        for(int i1: l1){
            for(int i2: l2){
                result = Math.min(result, Math.abs(i1-i2));
            }
        }
        return result;
    }
}
```

---

The time complexity for shortest method is O(M\*N), where M is frequency of word<sub>1</sub> and N is the frequency of word<sub>2</sub>. This can be improved by the following:

---

```
public int shortest(String word1, String word2) {

    ArrayList<Integer> l1 = map.get(word1);
    ArrayList<Integer> l2 = map.get(word2);

    int result = Integer.MAX_VALUE;
```

---

```
int i=0;
int j=0;
while(i<l1.size() && j<l2.size()){
    result = Math.min(result, Math.abs(l1.get(i)-l2.get(j)));
    if(l1.get(i)<l2.get(j)){
        i++;
    }else{
        j++;
    }
}

return result;
}
```

---

The time complexity of the shortest method is now  $O(M+N)$ . Since  $M+N < \text{size of word list}$ , the time is  $O(K)$  where  $k$  is the list size.

# 20 Shortest Word Distance III

This is a follow-up problem of [Shortest Word Distance](#). The only difference is now word<sub>1</sub> could be the same as word<sub>2</sub>.

Given a list of words and two words word<sub>1</sub> and word<sub>2</sub>, return the shortest distance between these two words in the list.

word<sub>1</sub> and word<sub>2</sub> may be the same and they represent two individual words in the list.

For example, Assume that words = ["practice", "makes", "perfect", "coding", "makes"].

Given word<sub>1</sub> = "makes", word<sub>2</sub> = "coding", return 1. Given word<sub>1</sub> = "makes", word<sub>2</sub> = "makes", return 3.

## 20.1 Java Solution 1

In this problem, word<sub>1</sub> and word<sub>2</sub> can be the same. The two variables used to track indices should take turns to update.

---

```
public int shortestWordDistance(String[] words, String word1, String word2) {
    if(words==null || words.length<1 || word1==null || word2==null)
        return 0;

    int m=-1;
    int n=-1;

    int min = Integer.MAX_VALUE;
    int turn=0;
    if(word1.equals(word2))
        turn = 1;

    for(int i=0; i<words.length; i++){
        String s = words[i];
        if(word1.equals(s) && (turn ==1 || turn==0)){
            m = i;
            if(turn==1) turn=2;
            if(n!=-1)
                min = Math.min(min, m-n);
        }else if(word2.equals(s) && (turn==2 || turn==0)){
            n = i;
            if(turn==2) turn =1;
            if(m!=-1)
                min = Math.min(min, n-m);
        }
    }

    return min;
}
```

---

## 20.2 Java Solution 2

We can divide the cases to two: word<sub>1</sub> and word<sub>2</sub> are the same and not the same.

---

```
public int shortestWordDistance(String[] words, String word1, String word2) {
    if(words==null||words.length==0)
        return -1;

    if(word1==null || word2==null)
        return -1;

    boolean isSame = false;

    if(word1.equals(word2))
        isSame = true;

    int shortest= Integer.MAX_VALUE;

    int prev=-1;
    int i1=-1;
    int i2=-1;

    for(int i=0; i<words.length; i++){
        if(isSame){
            if(words[i].equals(word1)){
                if(prev!=-1){
                    shortest=Math.min(shortest, i-prev);
                }
                prev = i;
            }
        }else{
            if(word1.equals(words[i])){
                i1=i;
                if(i2!=-1){
                    shortest = Math.min(shortest, i-i2);
                }
            }else if(word2.equals(words[i])){
                i2=i;
                if(i1!=-1){
                    shortest = Math.min(shortest, i-i1);
                }
            }
        }
    }

    return shortest;
}
```

---

# 21 Intersection of Two Arrays

Given two arrays, write a function to compute their intersection.

## 21.1 Java Solution 1 - HashSet

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

---

```
public int[] intersection(int[] nums1, int[] nums2) {
    HashSet<Integer> set1 = new HashSet<Integer>();
    for(int i: nums1){
        set1.add(i);
    }

    HashSet<Integer> set2 = new HashSet<Integer>();
    for(int i: nums2){
        if(set1.contains(i)){
            set2.add(i);
        }
    }

    int[] result = new int[set2.size()];
    int i=0;
    for(int n: set2){
        result[i++] = n;
    }

    return result;
}
```

---

## 21.2 Java Solution 2 - Binary Search

Time =  $O(n \log(n))$ . Space =  $O(n)$ .

---

```
public int[] intersection(int[] nums1, int[] nums2) {
    Arrays.sort(nums1);
    Arrays.sort(nums2);

    ArrayList<Integer> list = new ArrayList<Integer>();
    for(int i=0; i<nums1.length; i++){
        if(i==0 || (i>0 && nums1[i]!=nums1[i-1])){
            if((Arrays.binarySearch(nums2, nums1[i])>-1){
                list.add(nums1[i]);
            }
        }
    }

    int[] result = new int[list.size()];
    int k=0;
```

```
for(int i: list){  
    result[k++] = i;  
}  
  
return result;  
}
```

---

### 21.3 Any improvement?

## 22 Intersection of Two Arrays II

Given two arrays, write a function to compute their intersection.

Example: Given  $\text{nums1} = [1, 2, 2, 1]$ ,  $\text{nums2} = [2, 2]$ , return  $[2, 2]$ .

### 22.1 Java Solution 1

---

```
public int[] intersect(int[] nums1, int[] nums2) {
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    for(int i: nums1){
        if(map.containsKey(i)){
            map.put(i, map.get(i)+1);
        }else{
            map.put(i, 1);
        }
    }

    ArrayList<Integer> list = new ArrayList<Integer>();
    for(int i: nums2){
        if(map.containsKey(i)){
            if(map.get(i)>1){
                map.put(i, map.get(i)-1);
            }else{
                map.remove(i);
            }
            list.add(i);
        }
    }

    int[] result = new int[list.size()];
    int i=0;
    while(i<list.size()){
        result[i]=list.get(i);
        i++;
    }

    return result;
}
```

---

### 22.2 Java Solution 2

If the arrays are sorted, then we can use two points.

---

```
public int[] intersect(int[] nums1, int[] nums2) {
    Arrays.sort(nums1);
    Arrays.sort(nums2);
    ArrayList<Integer> list = new ArrayList<Integer>();
    int p1=0, p2=0;
```

```
while(p1<nums1.length && p2<nums2.length){  
    if(nums1[p1]<nums2[p2]){  
        p1++;  
    }else if(nums1[p1]>nums2[p2]){  
        p2++;  
    }else{  
        list.add(nums1[p1]);  
        p1++;  
        p2++;  
    }  
}  
  
int[] result = new int[list.size()];  
int i=0;  
while(i<list.size()){  
    result[i]=list.get(i);  
    i++;  
}  
return result;  
}
```

---

## 23 Two Sum II Input array is sorted

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

To solve this problem, we can use two pointers 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;
}
```

---

## 24 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
```

---

### 24.1 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;
    }
}
```

---

# 25 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)

---

## 25.1 Java Solution

This problem can be solved by using two pointers. Time complexity is  $O(n^2)$ .

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

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

    if(nums == null || nums.length<3)
        return result;

    Arrays.sort(nums);

    for(int i=0; i<nums.length-2; i++){
        if(i==0 || nums[i] > nums[i-1]){
            int j=i+1;
            int k=nums.length-1;

            while(j<k){
                if(nums[i]+nums[j]+nums[k]==0){
                    List<Integer> l = new ArrayList<Integer>();
                    l.add(nums[i]);
                    l.add(nums[j]);
                    l.add(nums[k]);
                    result.add(l);

                    j++;
                    k--;
                }

                //handle duplicate here
                while(j<k && nums[j]==nums[j-1])
                    j++;
                while(j<k && nums[k]==nums[k+1])
                    k--;
            }
        }
    }
}
```

```
        }else if(nums[i]+nums[j]+nums[k]<0){
            j++;
        }else{
            k--;
        }
    }

    return result;
}
```

---

## 26 4Sum

Given an array S of n integers, are there elements a, b, c, and d in S such that  $a + b + c + d = \text{target}$ ? 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,  $a \leq b \leq c \leq d$ ) 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)

---

### 26.1 Java Solution

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

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

    if(nums==null || nums.length<4)
        return result;

    Arrays.sort(nums);

    for(int i=0; i<nums.length-3; i++){
        if(i!=0 && nums[i]==nums[i-1])
            continue;
        for(int j=i+1; j<nums.length-2; j++){
            if(j!=i+1 && nums[j]==nums[j-1])
                continue;
            int k=j+1;
            int l=nums.length-1;
            while(k<l){
                if(nums[i]+nums[j]+nums[k]+nums[l]<target){
                    k++;
                }else if(nums[i]+nums[j]+nums[k]+nums[l]>target){
                    l--;
                }else{
                    List<Integer> t = new ArrayList<Integer>();
                    t.add(nums[i]);
                    t.add(nums[j]);
                    t.add(nums[k]);
                    t.add(nums[l]);
                    result.add(t);
                }
            }
        }
    }
}
```

```
while(k<l &&nums[l]==nums[l+1] ){
    l--;
}

while(k<l &&nums[k]==nums[k-1]){
    k++;
}
}

return result;
}
```

---

# 27 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).

---

## 27.1 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.

## 27.2 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--;
            }
        }
    }

    return result;
}
```

---

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

# 28 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
```

---

## 28.1 Java Solution

This is a binary search problem. The complexity should be  $O(\log(n))$ .

---

```
public int searchInsert(int[] nums, int target) {
    if(target>nums[nums.length-1]){
        return nums.length;
    }

    int l=0;
    int r=nums.length-1;

    while(l<r){
        int m = l+(r-l)/2;
        if(target>nums[m]){
            l=m+1;
        }else{
            r=m;
        }
    }

    return l;
}
```

---

Or similarly, we can write the solution like the following:

---

```
public int searchInsert(int[] nums, int target) {
    int i=0;
    int j=nums.length-1;

    while(i<=j){
        int mid = (i+j)/2;

        if(target > nums[mid]){
            i=mid+1;
        }else if(target < nums[mid]){
            j=mid-1;
        }else{
            return mid;
        }
    }

    return i;
}
```

---

```
    }
}

return i;
}
```

---

# 29 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  $O(\log(m+n))$ .

## 29.1 Java Solution

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

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

For normal cases(all other cases), we need to move the pointer at the pace of half of the array size to get  $O(\log(n))$  time.

```
public double findMedianSortedArrays(int[] nums1, int[] nums2) {
    int total = nums1.length+nums2.length;
    if(total%2==0){
        return (getKth(nums1, 0, nums1.length-1, nums2, 0, nums2.length-1, total/2)
            + getKth(nums1, 0, nums1.length-1, nums2, 0, nums2.length-1, total/2-1))/2.0;
    }else{
        return getKth(nums1,0, nums1.length-1, nums2, 0, nums2.length-1, total/2);
    }
}

//k is the index starting from 0
private int getKth(int[] nums1, int i1, int j1, int[] nums2, int i2, int j2, int k){
    if(j1<i1){
        return nums2[i2+k];
    }
    if(j2<i2){
        return nums1[i1+k];
    }

    if(k==0){
        return Math.min(nums1[i1], nums2[i2]);
    }

    int len1 = j1 - i1 + 1;
    int len2 = j2 - i2 + 1;

    int m1 = k*len1/(len1+len2);
    int m2 = k - m1 - 1;

    m1 += i1;
    m2 += i2;

    if(nums1[m1]<nums2[m2]){
        k = k-(m1-i1+1);
        j2 = m2;
        i1 = m1+1;
    }else{
        k = k-(m2-i2+1);
    }
}
```

```
j1 = m1;
i2 = m2+1;
}

return getKth(nums1, i1, j1, nums2, i2, j2, k);
}
```

The main challenge is to calculate the middle elements, we can not do the following like a regular binary search:

```
int m1 = i1+(j1-i1)/2;
int m2 = i2+(j2-i2)/2;
```

It will result in either dead loop or missing the element at the beginning. The key is we always drop  $\leq$  half size of the elements.

# 30 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  $4 \ 5 \ 6 \ 7 \ 0 \ 1 \ 2$ ).

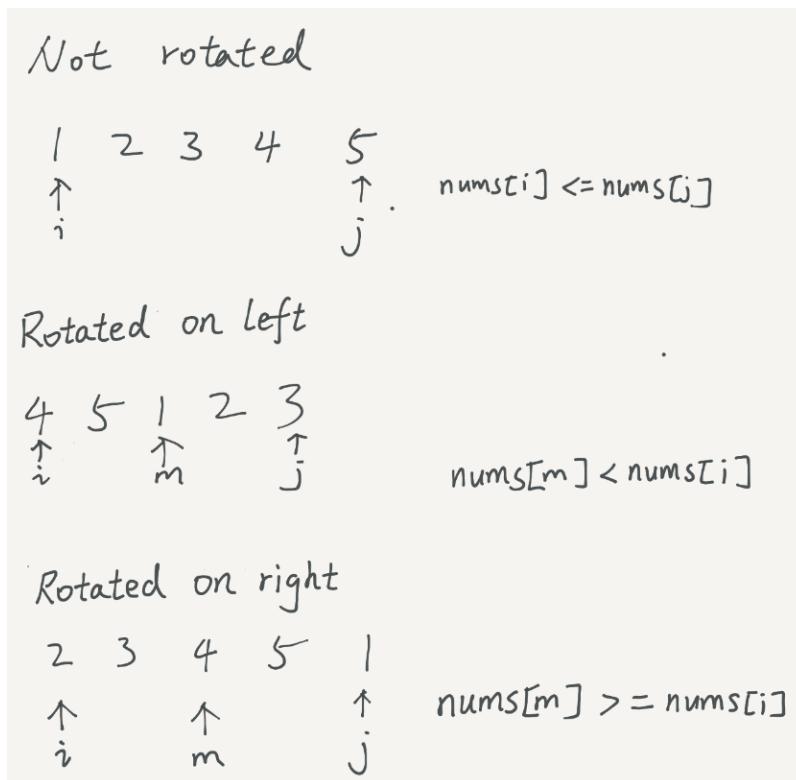
Find the minimum element. You may assume no duplicate exists in the array.

## 30.1 Analysis

This problem is a binary search and the key is breaking the array to two parts, so that we can 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.



## 30.2 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);
    }
}

```

---

### 30.3 Java Solution 2 - Iteration

---

```

/*
To understand the boundaries, use the following 3 examples:
[2,1], [2,3,1], [3,1,2]
*/
public int findMin(int[] nums) {
    if(nums==null || nums.length==0)
        return -1;

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

    while(i<=j){
        if(nums[i]<=nums[j])
            return nums[i];

        int m=(i+j)/2;

        if(nums[m]>=nums[i]){
            i=m+1;
        }else{
            j=m;
        }
    }

    return -1;
}

```

---

# 31 Find Minimum in Rotated Sorted Array II

Follow up for "Find Minimum in Rotated Sorted Array": What if duplicates are allowed?  
Would this affect the run-time complexity? How and why?

## 31.1 Java Solution 1 - Recursion

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

---

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

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

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

---

## 31.2 Java Solution 2 - Iteration

---

```
public int findMin(int[] nums) {
    int i=0;
    int j=nums.length-1;

    while(i<=j){
```

```
//handle cases like [3, 1, 3]
while(nums[i]==nums[j] && i!=j){
    i++;
}

if(nums[i]<=nums[j]){
    return nums[i];
}

int m=(i+j)/2;
if(nums[m]>=nums[i]){
    i=m+1;
}else{
    j=m;
}

return -1;
}
```

---

## 32 Find First and Last Position of Element in Sorted Array

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  $[3, 4]$ .

### 32.1 Analysis

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

### 32.2 Java Solution 1 - Log(n)

We can first find the start and then the end of the target.

---

```
public int[] searchRange(int[] nums, int target) {
    int l=0;
    int r=nums.length-1;

    while(l<r){
        int m=l+(r-l)/2;
        if(nums[m]<target){
            l=m+1;
        }else{
            r=m;
        }
    }

    int first=l;
    if(l<nums.length&&nums[l]==target){//l is in boundary and is the target
        l=0;
        r=nums.length-1;
        while(l<r){
            int m=l+(r-l+1)/2;
            if(nums[m]>target){
                r=m-1;
            }else{
                l=m;
            }
        }

        return new int[]{first, r};
    }

    return new int[]{-1,-1};
}
```

---

### 32.3 Java Solution 2 - (Deprecated)

---

```

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;

    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;
        }
    }
}

```

---

In the worst case, the time of the second solution is actually  $O(n)$ .

# 33 Guess Number Higher or Lower

We are playing the Guess Game. The game is as follows:

I pick a number from 1 to n. You have to guess which number I picked.

Every time you guess wrong, I'll tell you whether the number is higher or lower.

You call a pre-defined API `guess(int num)` which returns 3 possible results (-1, 1, or 0):

-1 : My number is lower 1 : My number is higher 0 : Congrats! You got it! Example: n = 10, I pick 6.

Return 6.

## 33.1 Java Solution

This is a typical binary search problem. Here is a Java solution.

---

```
public int guessNumber(int n) {
    int low=1;
    int high=n;

    while(low <= high){
        int mid = low+((high-low)/2);
        int result = guess(mid);
        if(result==0){
            return mid;
        }else if(result==1){
            low = mid+1;
        }else{
            high=mid-1;
        }
    }

    return -1;
}
```

---

## 33.2 What we learn from this problem?

$\text{low} + (\text{high}-\text{low})/2$  yields the same value with  $(\text{low}+\text{high})/2$ . However, the first expression is less expensive. In addition, the following expression can be used:

---

```
low+((high-low)>>1)
(low+high)>>>1
```

---

Under the assumption that high and low are both non-negative, we know for sure that the upper-most bit (the sign-bit) is zero.

So both high and low are in fact 31-bit integers.

---

```
high = 0100 0000 0000 0000 0000 0000 0000 = 1073741824
low = 0100 0000 0000 0000 0000 0000 0000 = 1073741824
```

---

When you add them together they may "spill" over into the top-bit.

```
high + low = 1000 0000 0000 0000 0000 0000 0000 0000  
= 2147483648 as unsigned 32-bit integer  
= -2147483648 as signed 32-bit integer
```

```
(high + low) / 2 = 1100 0000 0000 0000 0000 0000 0000 0000 = -1073741824  
(high + low) >> 1 = 0100 0000 0000 0000 0000 0000 0000 0000 = 1073741824
```

---

## 34 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.

### 34.1 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
    }
}
```

---

### 34.2 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;
        } else {
            i = m+1;
        }
    }

    if (isBadVersion(i)) {
        return i;
    }

    return j;
}
```

}

---

# 35 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.

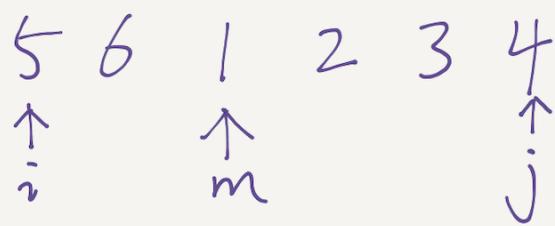
## 35.1 Analysis

In order to use binary search on the rotated sorted array, we need to determine how to update the left and right pointers. There are two major cases as shown below:

Case 1:  $a[m] > a[i]$   $\rightarrow$  shift point is on right side



Case 2:  $a[m] < a[i]$   $\rightarrow$  shift point is on left side



Once the two cases are identified, the problem is straightforward to solve. We only need to check if the target element is in the sorted side, and based on that move left or right pointers.

## 35.2 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);
    }
}
}
}

```

---

### 35.3 Java Solution 2 - Iterative

```

public int search(int[] nums, int target) {
    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;
}

```

---

# 36 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.

## 36.1 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;
}
```

---

# 37 Longest Increasing Subsequence

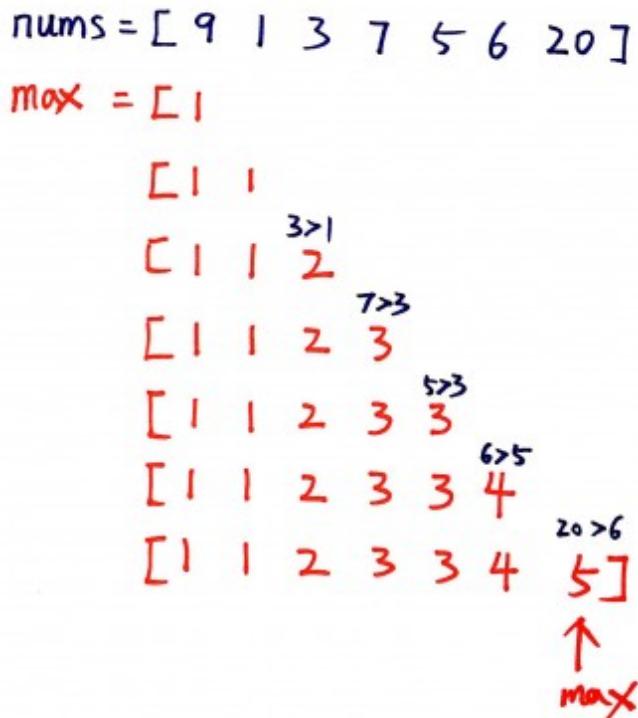
Given an unsorted array of integers, find the length of longest increasing subsequence.

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.

## 37.1 Java Solution 1 - Naive

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

Here is an example:



---

```
public int lengthOfLIS(int[] nums) {  
    if(nums==null || nums.length==0)  
        return 0;  
  
    int[] max = new int[nums.length];  
  
    for(int i=0; i<nums.length; i++){  
        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;
}

```

---

Or, simplify the code as:

---

```

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

    int[] max = new int[nums.length];
    Arrays.fill(max, 1);

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

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

    return result;
}

```

---

## 37.2 Java Solution 2 - Binary Search

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 | 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 || 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();
}

```

Note that the problem asks the length of the sequence, not the sequence itself.

# 38 Count of Smaller Numbers After Self

You are given an integer array `nums` and you have to return a new `counts` array. The `counts` array has the property where `counts[i]` is the number of smaller elements to the right of `nums[i]`.

Example:

Input: [5,2,6,1] Output: [2,1,1,0]

## 38.1 Java Solution 1

---

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

    for(int i=nums.length-1; i>=0; i--){
        if(sorted.isEmpty()){
            sorted.add(nums[i]);
            result.add(0);
        }else if(nums[i]>sorted.get(sorted.size()-1)){
            sorted.add(sorted.size(), nums[i]);
            result.add(sorted.size()-1);
        }else{
            int l=0;
            int r=sorted.size()-1;

            while(l<r){
                int m = l + (r-l)/2;

                if(nums[i]>sorted.get(m)){
                    l=m+1;
                }else{
                    r=m;
                }
            }

            sorted.add(r, nums[i]);
            result.add(r);
        }
    }

    Collections.reverse(result);

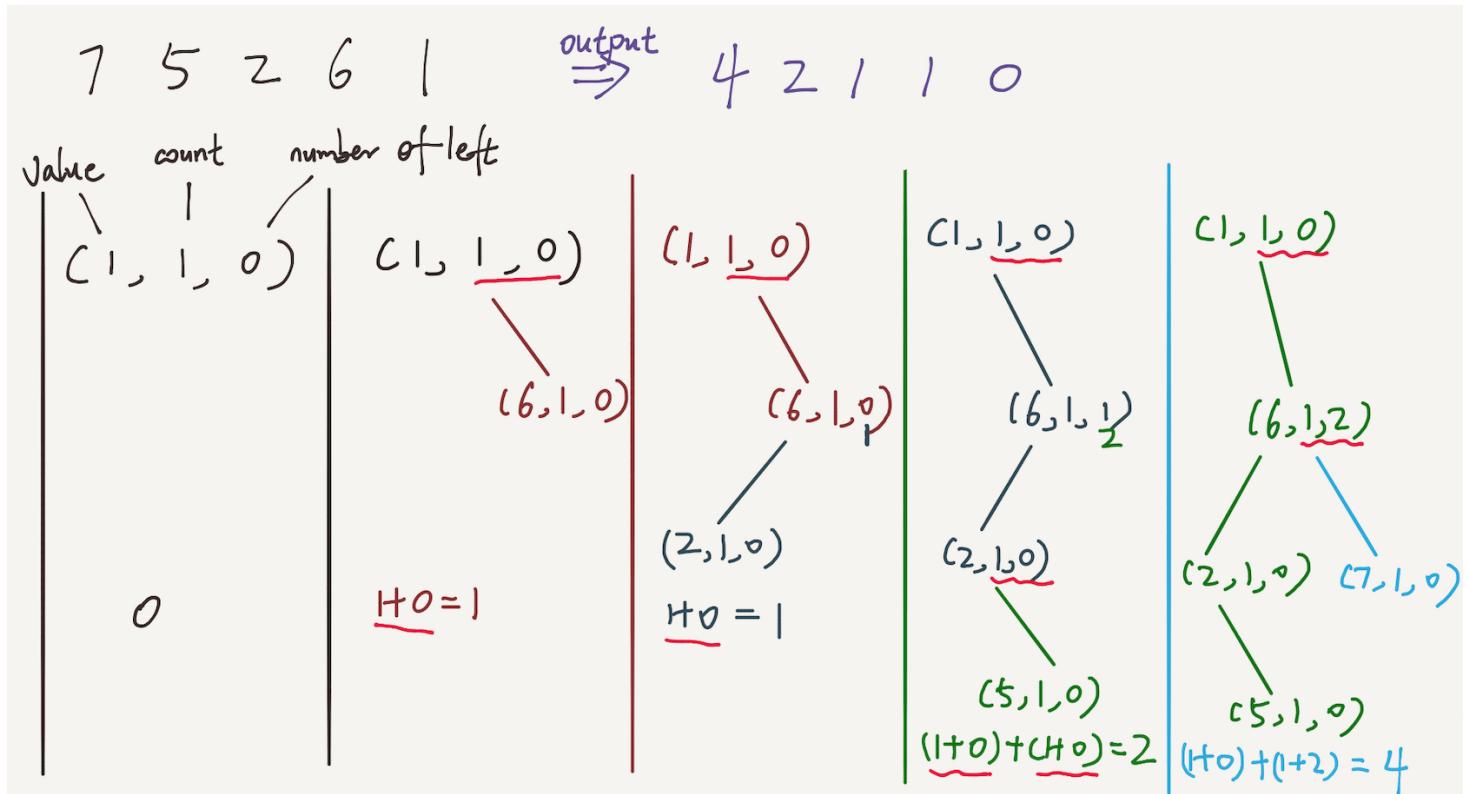
    return result;
}
```

---

This solution is simple. However, note that time complexity of adding an element to a list is  $O(n)$ , because elements after the insertion position need to be shifted. So the time complexity is  $O(n^2 \log n)$ .

## 38.2 Java Solution 2

If we want to use binary search, and define a structure like the following:



On average, time complexity is  $O(n \cdot \log(n))$  and space complexity is  $O(n)$ .

```

class Solution {
    public List<Integer> countSmaller(int[] nums) {

        List<Integer> result = new ArrayList<Integer>();
        if(nums==null || nums.length==0){
            return result;
        }

        Node root = new Node(nums[nums.length-1]);
        root.count=1;
        result.add(0);

        for(int i=nums.length-2; i>=0; i--){
            result.add(insertNode(root, nums[i]));
        }

        Collections.reverse(result);

        return result;
    }

    public int insertNode(Node root, int value){
        Node p=root;
        int result=0;
    }
}

```

```
while(p!=null){
    if(value>p.value){
        result+=p.count+p.numLeft;
        if(p.right==null){
            Node t = new Node(value);
            t.count=1;
            p.right=t;
            return result;
        }else{
            p=p.right;
        }
    }else if(value==p.value){
        p.count++;
        return result+p.numLeft;
    }else{
        p.numLeft++;

        if(p.left==null){
            Node t = new Node(value);
            t.count=1;
            p.left=t;
            return result;
        }else{
            p=p.left;
        }
    }
}

return 0;
}
}

class Node{
    Node left;
    Node right;

    int value;
    int count;
    int numLeft;
    public Node(int value){
        this.value=value;
    }
}
```

# 39 Russian Doll Envelopes

You have a number of envelopes with widths and heights given as a pair of integers (w, h). One envelope can fit into another if and only if both the width and height of one envelope is greater than the width and height of the other envelope.

What is the maximum number of envelopes can you Russian doll? (put one inside other)

## 39.1 Java Solution 1 - Naive

---

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

    Arrays.sort(envelopes, new Comparator<int[]>(){
        public int compare(int[] a, int[] b){
            if(a[0]!=b[0]){
                return a[0]-b[0];
            }else{
                return a[1]-b[1];
            }
        }
    });
    int max=1;
    int[] arr = new int[envelopes.length];
    for(int i=0; i<envelopes.length; i++){
        arr[i]=1;
        for(int j=i-1; j>=0; j--){
            if(envelopes[i][0]>envelopes[j][0]&&envelopes[i][1]>envelopes[j][1]){
                arr[i]=Math.max(arr[i], arr[j]+1);
            }
        }
        max = Math.max(max, arr[i]);
    }

    return max;
}
```

---

## 39.2 Java Solution 2 - Binary Search

We can sort the envelopes by height in ascending order and width in descending order. Then look at the width and find the longest increasing subsequence. This problem is then converted to the problem of finding [Longest Increasing Subsequence](#).

---

```
public int maxEnvelopes(int[][] envelopes) {
    Comparator c = Comparator.comparing((int[] arr) -> arr[0])
        .thenComparing((int[] arr) -> arr[1], Comparator.reverseOrder());
    Arrays.sort(envelopes, c);
```

---

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

for(int[] arr: envelopes){
    int target = arr[1];

    if(list.isEmpty()||target>list.get(list.size()-1)){
        list.add(target);
    }else{
        int i=0;
        int j=list.size()-1;

        while(i<j){
            int m = i + (j-i)/2;
            if(list.get(m)>=target){
                j = m;
            }else{
                i = m+1;
            }
        }

        list.set(j, target);
    }
}

return list.size();
}
```

---

Time complexity is  $O(n \cdot \log(n))$  and we need  $O(n)$  of space for the list.

## 40 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.

### 40.1 Java Solution 1

idx	0	1	2	3	4	5
	0	1	3	5	5	6

$i=5$ , count = 1, citation = 6,  $6 \geq 1$ , update h-index, continue  
 $i=4$ , count = 2, citation = 5,  $5 \geq 2$ , update h-index, continue  
 $i=3$ , count = 3, citation = 5,  $5 \geq 3$ , update h-index, continue  
 $i=2$ , count = 4, citation = 3,  $3 \not\geq 4$ , break.  
Result = 3.

---

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

    int result = 0;
    for(int i=citations.length-1; i>=0; i--){
        int cnt = citations.length-i;
        if(citations[i]>=cnt){
            result = cnt;
        }else{
            break;
        }
    }

    return result;
}
```

---

## 40.2 Java Solution 2

We can also put the citations in a counting sort array, then iterate over the counter array.

---

```
public int hIndex(int[] citations) {
    int len = citations.length;
    int[] counter = new int[len+1];

    for(int c: citations){
        counter[Math.min(len,c)]++;
    }

    int k=len;
    for(int s=counter[len]; k > s; s += counter[k]){
        k--;
    }

    return k;
}
```

---

# 41 HIndex II

Follow up for H-Index: What if the citations array is sorted in ascending order? Could you optimize your algorithm?

## 41.1 Java Solution

Given the array is sorted, we should use binary search.

---

```
int hIndex(int[] citations) {
    int len = citations.length;

    if (len == 0) {
        return 0;
    }

    if (len == 1) {
        if (citations[0] == 0) {
            return 0;
        } else {
            return 1;
        }
    }

    int i = 0;
    int j = len - 1;
    while (i < j) {
        int m = i + (j - i + 1) / 2;
        if (citations[m] > len - m) {
            j = m - 1;
        } else {
            i = m;
        }
    }

    if (citations[j] > len - j) {
        return len;
    }

    if (citations[j] == len - j) {
        return len - j;
    } else {
        return len - j - 1;
    }
}
```

---

## 42 Valid Anagram

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

### 42.1 Java Solution 1

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

---

```
public boolean isAnagram(String s, String t) {  
    if(s==null || t==null)  
        return false;  
  
    if(s.length()!=t.length())  
        return false;  
  
    int[] arr = new int[26];  
    for(int i=0; i<s.length(); i++){  
        arr[s.charAt(i)-'a']++;  
        arr[t.charAt(i)-'a']--;  
    }  
  
    for(int i: arr){  
        if(i!=0)  
            return false;  
    }  
  
    return true;  
}
```

---

### 42.2 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)){  
            map.put(c2, map.get(c2)-1);  
        }else{  
            map.put(c2,-1);  
        }  
    }  
  
    for(Map.Entry<Character, Integer> entry : map.entrySet()){  
        if(entry.getValue() != 0)  
            return false;  
    }  
    return true;  
}
```

---

```
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;
}
```

---

## 43 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"]  
]
```

### 43.1 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);  
        }  
    }  
  
    for(Map.Entry<String, ArrayList<String>> entry: map.entrySet()){  
        Collections.sort(entry.getValue());  
    }  
}
```

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

---

## 44 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. words[i] + words[j] is a palindrome.

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

### 44.1 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);
            }
        }
    }

    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;
}

```

# 45 Line Reflection

Given n points on a 2D plane, find if there is such a line parallel to y-axis that reflects the given points.

Example 1: Given points = [[1,1],[-1,1]], return true.

Example 2: Given points = [[1,1],[-1,-1]], return false.

Follow up: Could you do better than O(n<sup>2</sup>)?

## 45.1 Java Solution

For this problem, we first find the smallest and largest x-value for all points and get the line's x-axis is (minX + maxX) / 2, then for each point, check if each point has a reflection points in the set.

```
public boolean isReflected(int[][] points) {
    if(points==null || points.length<2)
        return true;

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

    int min=Integer.MAX_VALUE;
    int max=Integer.MIN_VALUE;

    for(int[] arr: points){
        min = Math.min(min, arr[0]);
        max = Math.max(max, arr[0]);
        HashSet<Integer> set = map.get(arr[0]);
        if(set==null){
            set = new HashSet<Integer>();
            map.put(arr[0], set);
        }
        set.add(arr[1]);
    }

    int y = min+max;

    for(int[] arr: points){
        int left = arr[0];
        int right = y-left;
        if(map.get(right)==null || !map.get(right).contains(arr[1])){
            return false;
        }
    }
    return true;
}
```

# 46 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.

## 46.1 Analysis

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

## 46.2 Java Solution

---

```
public boolean isIsomorphic(String s, String t) {
    if(s.length()!=t.length()){
        return false;
    }

    HashMap<Character, Character> map1 = new HashMap<>();
    HashMap<Character, Character> map2 = new HashMap<>();

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

        if(map1.containsKey(c1)){
            if(c2!=map1.get(c1)){
                return false;
            }
        } else{
            if(map2.containsKey(c2)){
                return false;
            }

            map1.put(c1, c2);
            map2.put(c2, c1);
        }
    }

    return true;
}
```

---

Time complexity is O(n) and space complexity is O(n), where n is the length of the input string.

## 47 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=0, index2=1

---

### 47.1 Java Solution

The optimal solution to solve this problem is using a HashMap. For each element of the array, (target-nums[i]) and the index are stored in the HashMap.

---

```
public int[] twoSum(int[] nums, int target) {
    if(nums==null || nums.length<2)
        return new int[]{0,0};

    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    for(int i=0; i<nums.length; i++){
        if(map.containsKey(nums[i])){
            return new int[]{map.get(nums[i]), i};
        }else{
            map.put(target-nums[i], i);
        }
    }

    return new int[]{0,0};
}
```

---

Time complexity is O(n).

# 48 Maximum Size Subarray Sum Equals k

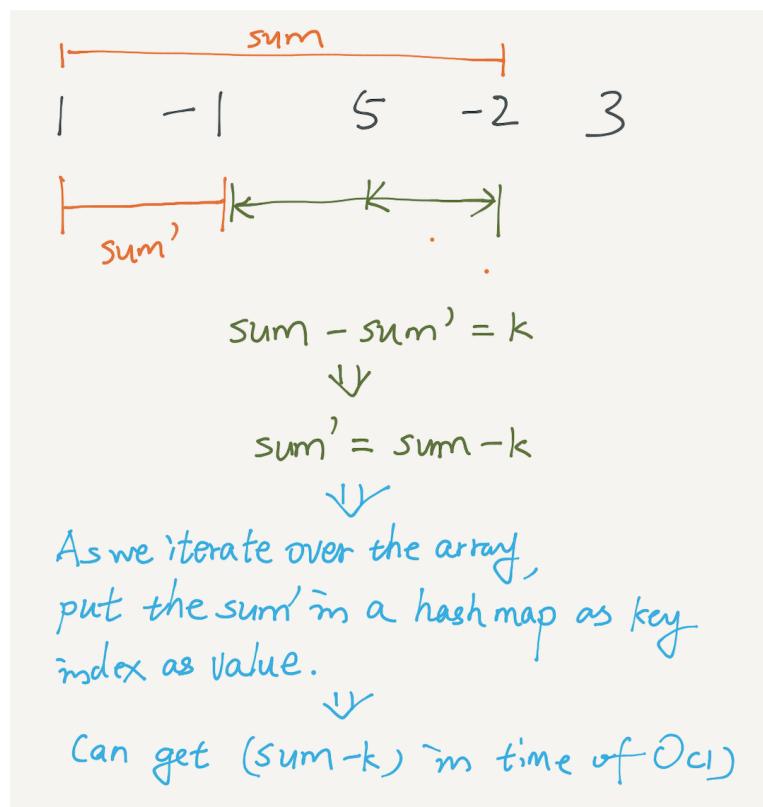
Given an array `nums` and a target value `k`, find the maximum length of a subarray that sums to `k`. If there isn't one, return 0 instead.

Note: The sum of the entire `nums` array is guaranteed to fit within the 32-bit signed integer range.

Example 1: Given `nums = [1, -1, 5, -2, 3]`, `k = 3`, return 4. (because the subarray `[1, -1, 5, -2]` sums to 3 and is the longest)

## 48.1 Java Solution

This problem is similar to Maximum Sum of SubArray Close to K.



```
public int maxSubArrayLen(int[] nums, int k) {
    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
    int max = 0;
    int sum=0;
    for(int i=0; i<nums.length; i++){
        sum += nums[i];
        if(sum==k){
            max = i+1;
        }
        if(map.containsKey(sum-k)){
            max = Math.max(max, i-map.get(sum-k));
        }
        if(!map.containsKey(sum)){
            map.put(sum, i);
        }
    }
    return max;
}
```

```
    max = Math.max(max, i+1);
}

int diff = sum-k;

if(map.containsKey(diff)){
    max = Math.max(max, i-map.get(diff));
}

if(!map.containsKey(sum)){
    map.put(sum, i);
}

return max;
}
```

---

# 49 Subarray Sum Equals K

Given an array of integers and an integer k, find the total number of continuous subarrays whose sum equals to k.

Example 1:

Input:nums = [1,1,1], k = 2 Output: 2

Note that empty array is not considered as a subarray.

## 49.1 Java Solution

The sum problems can often be solved by using a hash map efficiently.

---

```
public int subarraySum(int[] nums, int k) {
    HashMap<Integer, Integer> map = new HashMap<>();
    map.put(0, 1);

    int count = 0;
    int sum = 0;

    //e.g., 1 1 2 1 1
    for(int i=0; i<nums.length; i++){
        sum += nums[i];
        int n = map.getOrDefault(sum-k, 0);
        count += n;

        map.put(sum, map.getOrDefault(sum, 0)+1);
    }

    return count;
}
```

---

# 50 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.

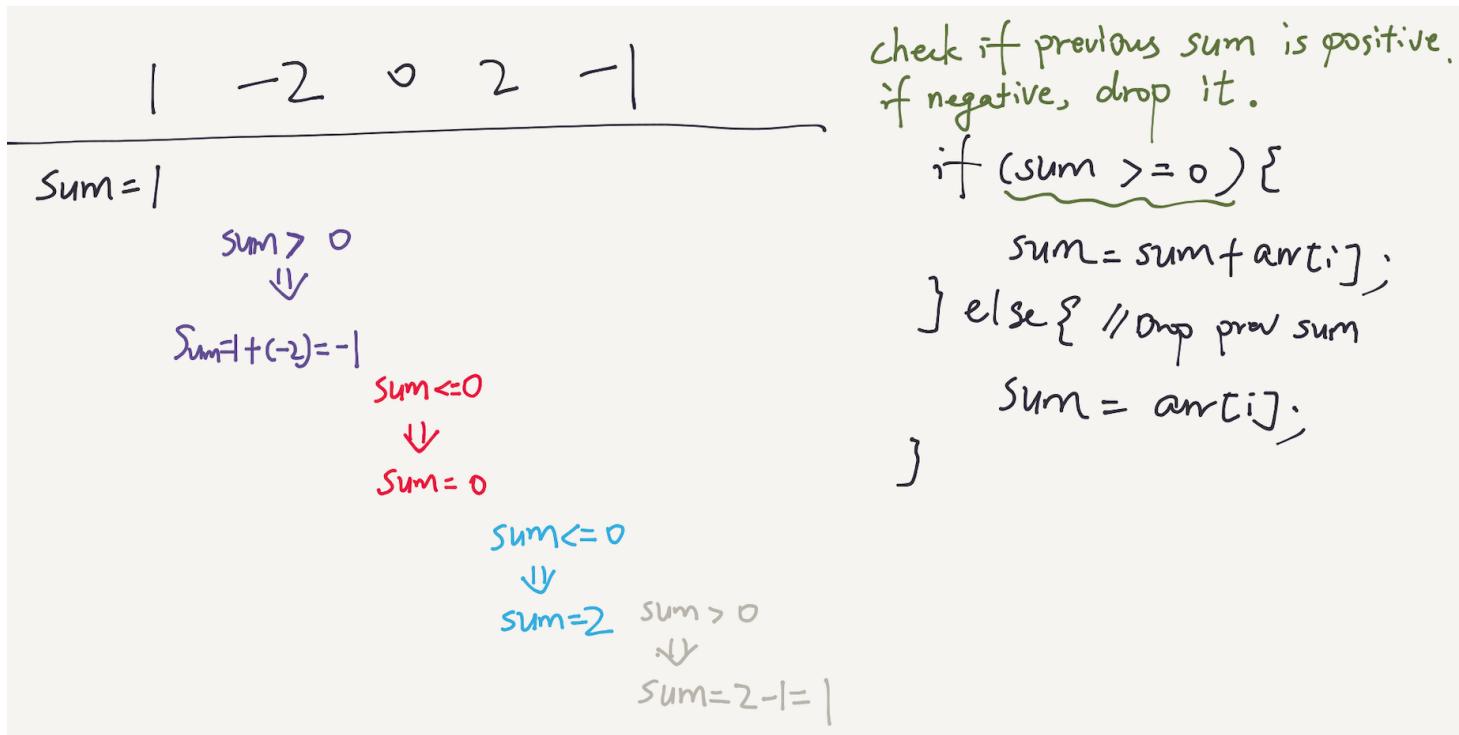
## 50.1 Java Solution 1 - DP

The easiest way to formulate the solution of this problem is using DP. Let  $f(n)$  be the maximum subarray for an array with  $n$  elements. We need to find the subproblem and the relation.

$$f(n) = \{ f(n-1) > 0 ? f(n-1) : 0 \} + \text{nums}[n-1]$$

$$f(0) = 0$$

$$f(1) = \text{nums}[0]$$



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;
    }
}

```

---

The time complexity and space complexity are the same  $O(n)$ . However, we can improve the space complexity and make it to be  $O(1)$ .

---

```

public int maxSubArray(int[] nums) {
    //[-2,1,-3,4,-1,2,1,-5,4],
    int result = nums[0];
    int sum = nums[0];
    for(int i=1; i<nums.length; i++){
        if(sum<=0){
            sum = nums[i];
        }else{
            sum += nums[i];
        }
        result = Math.max(result, sum);
    }

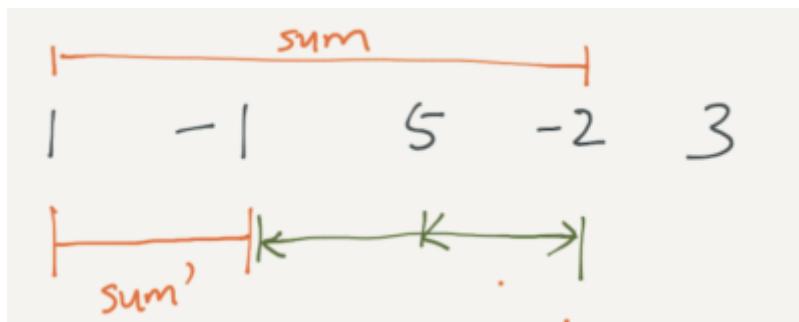
    return result;
}

```

---

## 50.2 Java Solution 2 - Using Sum Diff

Like other array sum related problems, the target sum  $k$  can be expressed as  $\text{sum} - \text{sum}'$ :




---

```

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

    PriorityQueue<Integer> pre= new PriorityQueue<>();
    pre.offer(0);

    int sum = 0;

```

```
int result = Integer.MIN_VALUE;

for(int i=0; i<nums.length; i++){
    sum = sum+nums[i];
    result = Math.max(result, sum-pre.peek());
    pre.offer(sum);
}

return result;
}
```

---

Time complexity is  $O(n\log n)$  and space complexity is  $O(n)$ .

# 51 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.

## 51.1 Java Solution - Dynamic Programming

This is similar to [maximum subarray](#). Instead of sum, the sign of number affect the product value.

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

```
public int maxProduct(int[] nums) {  
    int[] max = new int[nums.length];  
    int[] min = new int[nums.length];  
  
    max[0] = min[0] = nums[0];  
    int result = nums[0];  
  
    for(int i=1; i<nums.length; i++){  
        if(nums[i]>0){  
            max[i]=Math.max(nums[i], max[i-1]*nums[i]);  
            min[i]=Math.min(nums[i], min[i-1]*nums[i]);  
        }else{  
            max[i]=Math.max(nums[i], min[i-1]*nums[i]);  
            min[i]=Math.min(nums[i], max[i-1]*nums[i]);  
        }  
  
        result = Math.max(result, max[i]);  
    }  
  
    return result;  
}
```

Time is O(n).

# 52 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 "aaaaa" the longest substring is "a", with the length of 1.

## 52.1 Analysis

The basic idea to solve this problem is using an extra data structure to track the unique characters in a sliding window. Both an array and a hash set work for this purpose.

## 52.2 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;
        }
    }

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

    return result;
}
```

---

}

### 52.3 Java Solution 2 - HashSet

Using a HashSet can simplify the code a lot.

---

```
/*
    pwwkew
i |
j |

i |
j |

i |
j |

*/
public int lengthOfLongestSubstring(String s) {
    if(s==null||s.length()==0){
        return 0;
    }

    HashSet<Character> set = new HashSet<>();
    int result = 1;
    int i=0;
    for(int j=0; j<s.length(); j++){
        char c = s.charAt(j);
        if(!set.contains(c)){
            set.add(c);
            result = Math.max(result, set.size());
        }else{
            while(i<j){
                if(s.charAt(i)==c){
                    i++;
                    break;
                }

                set.remove(s.charAt(i));
                i++;
            }
        }
    }

    return result;
}
```

---

# 53 Longest Substring with At Most K Distinct Characters

This is a problem asked by Google.

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

## 53.1 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?

## 53.2 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 result = 0;
    int i=0;
    HashMap<Character, Integer> map = new HashMap<Character, Integer>();

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

        if(map.size()<=k){
            result = Math.max(result, j-i+1);
        }else{
            while(map.size()>k){
                char l = s.charAt(i);
                int count = map.get(l);
                if(count==1){
                    map.remove(l);
                }else{
                    map.put(l, map.get(l)-1);
                }
                i++;
            }
        }
    }

    return result;
}

```

---

Time is O(n).

# 54 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  $[0,9]$ .

## 54.1 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.

## 54.2 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 totoal qualified words so far

        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++;
        }

        if(count==words.length){
            result.add(start);
        }
    }
}
```

```
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;
}
```

# 55 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".

## 55.1 Java Solution

```
public String minWindow(String s, String t) {
    HashMap<Character, Integer> goal = new HashMap<>();
    int goalSize = t.length();
    int minLen = Integer.MAX_VALUE;
    String result = "";

    //target dictionary
    for(int k=0; k<t.length(); k++){
        goal.put(t.charAt(k), goal.getOrDefault(t.charAt(k), 0)+1);
    }

    int i=0;
    int total=0;
    HashMap<Character, Integer> map = new HashMap<>();
    for(int j=0; j<s.length(); j++){
        char c = s.charAt(j);
        if(!goal.containsKey(c)){
            continue;
        }

        //if c is a target character in the goal, and count is < goal, increase the total
        int count = map.getOrDefault(c, 0);
        if(count<goal.get(c)){
            total++;
        }

        map.put(c, count+1);

        //when total reaches the goal, trim from left until no more chars can be trimmed.
        if(total==goalSize){
            while(!goal.containsKey(s.charAt(i)) || map.get(s.charAt(i))>goal.get(s.charAt(i))){
                char pc = s.charAt(i);
                if(goal.containsKey(pc) && map.get(pc)>goal.get(pc)){
                    map.put(pc, map.get(pc)-1);
                }

                i++;
            }

            if(minLen>j-i+1){
                minLen = j-i+1;
                result = s.substring(i, j+1);
            }
        }
    }
}
```

```
        }
    }
}

return result;
}
```

---

## 56 Longest Substring with At Least K Repeating Characters

Find the length of the longest substring T of a given string (consists of lowercase letters only) such that every character in T appears no less than k times.

Example 1:

---

Input:

s = "aaabb", k = 3

Output:

3

The longest substring is "aaa", as 'a' is repeated 3 times.

---

### 56.1 Java Solution

This problem can be solved using DFS. When all chars in the input string occurs  $\geq k$ , return the length. But we first need to split the input string by using the characters whose occurrence  $< k$ .

```
public int longestSubstring(String s, int k) {
    HashMap<Character, Integer> counter = new HashMap<Character, Integer>();

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

        char c = s.charAt(i);
        if(counter.containsKey(c)){
            counter.put(c, counter.get(c)+1);
        }else{
            counter.put(c, 1);
        }

    }

    HashSet<Character> splitSet = new HashSet<Character>();
    for(char c: counter.keySet()){
        if(counter.get(c)<k){
            splitSet.add(c);
        }
    }

    if(splitSet.isEmpty()){
        return s.length();
    }

    int max = 0;
    int i=0, j=0;
    while(j<s.length()){
        char c = s.charAt(j);
        if(splitSet.contains(c)){


```

```
    if(j!=i){  
        max = Math.max(max, longestSubstring(s.substring(i, j), k));  
    }  
    i=j+1;  
}  
j++;  
}  
  
if(i!=j)  
    max = Math.max(max, longestSubstring(s.substring(i, j), k));  
  
return max;  
}
```

---

# 57 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.

## 57.1 Java Solution 1

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

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;
```

## 57.2 Java Solution 2

We can also project the arrays to a new array with length to be the largest element in the array. Then iterate over the array and get the longest consecutive sequence. If the largest number is very large, then the time complexity would be bad.

## 58 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.)

### 58.1 Java Solution 1 - Sorting

Assuming the majority exists and since the majority always takes more than half of space, the middle element is guaranteed to be the majority. Sorting array takes  $O(n\log(n))$ . So the time complexity of this solution is  $n\log(n)$ .

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

### 58.2 Java Solution 2 - Majority Vote Algorithm

This problem can be solved in time of  $O(n)$  with constant space complexity. The basic idea is that the majority element can negate all other element's count.

		2	3	2	2
Result = 0		..	3	..	2
count = 0		0	1	0	1

```
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;  
}
```

---

# 59 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.

## 59.1 Java Solution

This problem is similar to Majority Element I. Time =  $O(n)$  and Space =  $O(1)$ .

```
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){
        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;
}
```

# 60 Increasing Triplet Subsequence

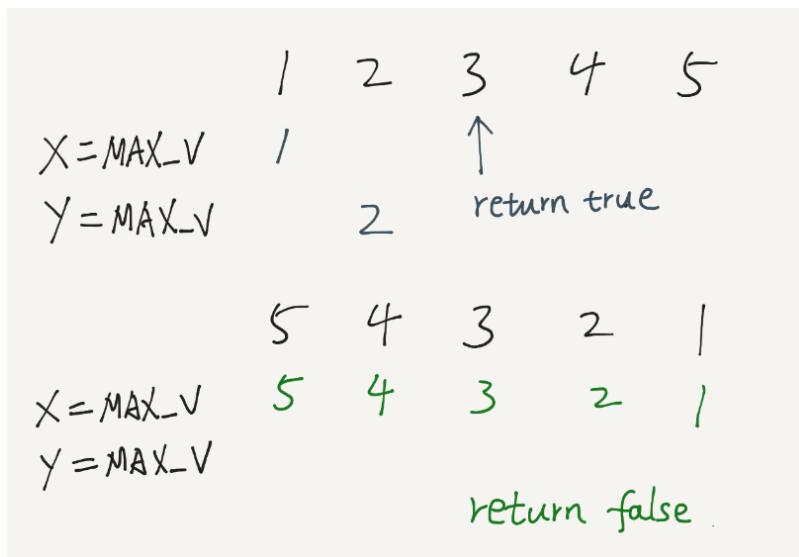
Given an unsorted array return whether an increasing subsequence of length 3 exists or not in the array.

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

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

## 60.1 Analysis

This problem can be formalized as finding a sequence x, y and z, such that  $x < y < z$ .



## 60.2 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;
}
```

}

---

# 61 Find the Second Largest Number in an Array

## 61.1 Problem

Given an integer array, find the second largest number in the array.

## 61.2 Solution

The optimal time is linear to the length of the array N. We can iterate over the array, whenever the largest elements get updated, its current value becomes the second largest.

---

```
public static int getSecondLargest(int[] arr){  
    int first = Integer.MIN_VALUE;  
    int second = Integer.MIN_VALUE;  
  
    for(int i=0; i<arr.length; i++){  
        if(arr[i]>first){  
            second=first;  
            first=arr[i];  
        }  
    }  
  
    return second;  
}
```

---

## 62 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.

### 62.1 Analysis

UPDATED on 06/07/2015.

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



### 62.2 Java Solution

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

---

# 63 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"]  
]
```

## 63.1 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.

## 63.2 Java Solution

```
class Solution {  
  
    public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {  
        List<List<String>> result = new ArrayList<List<String>>();  
  
        HashSet<String> unvisited = new HashSet<>();  
        unvisited.addAll(wordList);  
  
        LinkedList<Node> queue = new LinkedList<>();  
        Node node = new Node(beginWord, 0, null);  
        queue.offer(node);  
  
        int minLen = Integer.MAX_VALUE;  
        while(!queue.isEmpty()) {  
            Node top = queue.poll();  
  
            //top if have shorter result already  
            if(result.size() > 0 && top.depth > minLen){  
                return result;  
            }  
  
            for(int i=0; i < top.word.length(); i++) {  
                char c = top.word.charAt(i);  
                char[] arr = top.word.toCharArray();  
                for(char j='z'; j >= 'a'; j--) {  
                    if(j == c){  
                        continue;  
                    }  
                    arr[i] = j;  
                    String nextWord = new String(arr);  
                    if(nextWord.equals(endWord)) {  
                        List<String> ladder = new ArrayList<String>();  
                        ladder.add(beginWord);  
                        ladder.add(nextWord);  
                        result.add(ladder);  
                    } else if(unvisited.contains(nextWord)) {  
                        Node newNode = new Node(nextWord, top.depth + 1, top);  
                        queue.offer(newNode);  
                        unvisited.remove(nextWord);  
                    }  
                }  
            }  
        }  
        return result;  
    }  
}  
class Node {  
    String word;  
    int depth;  
    Node previous;  
}
```

```

        }
        arr[i]=j;
        String t = new String(arr);

        if(t.equals(endWord)){
            //add to result
            List<String> aResult = new ArrayList<>();
            aResult.add(endWord);
            Node p = top;
            while(p!=null){
                aResult.add(p.word);
                p = p.prev;
            }

            Collections.reverse(aResult);
            result.add(aResult);

            //stop if get shorter result
            if(top.depth<=minLen){
                minLen=top.depth;
            }else{
                return result;
            }
        }

        if(unvisited.contains(t)){
            Node n=new Node(t,top.depth+1,top);
            queue.offer(n);
            unvisited.remove(t);
        }
    }
}

return result;
}
}

class Node{
    public String word;
    public int depth;
    public Node prev;

    public Node(String word, int depth, Node prev){
        this.word=word;
        this.depth=depth;
        this.prev=prev;
    }
}

```

---

# 64 Top K Frequent Elements

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

## 64.1 Java Solution 1 - Heap

Time complexity is  $O(n * \log(k))$ . Note that heap is often used to reduce time complexity from  $n * \log(n)$  (see solution 3) to  $n * \log(k)$ .

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

class Solution {
    public List<Integer> topKFrequent(int[] nums, int k) {
        //count the frequency for each element
        HashMap<Integer, Integer> map = new HashMap<>();
        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<>(Comparator.comparing(Pair->Pair.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<>();
        while(queue.size()>0){
            result.add(queue.poll().num);
        }

        //reverse the order
        Collections.reverse(result);
    }
}
```

---

```

        return result;
    }
}

```

---

## 64.2 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 size==k, stop
                if(result.size()==k){
                    break;
                }
            }
        }
    }

    return result;
}

```

---

### 64.3 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;
        }

        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;
        }
    }
}

```

---

# 65 Meeting Rooms II

Given an array of meeting time intervals consisting of start and end times  $[[s_1, e_1], [s_2, e_2], \dots]$  find the minimum number of conference rooms required.

## 65.1 Java Solution

The basic idea of the solution is that we sequentially assign meeting to a room. We use a min heap to track the earliest ending meeting. Whenever an old meeting ends before a new meeting starts, we remove the old meeting. Otherwise, we need an extra room.

---

```
public int minMeetingRooms(Interval[] intervals) {
    if(intervals==null||intervals.length==0){
        return 0;
    }

    Comparator<Interval> comp = Comparator.comparing((Interval i)->i.start);
    Arrays.sort(intervals, comp);

    PriorityQueue<Integer> queue = new PriorityQueue<>();
    queue.offer(intervals[0].end);
    int count = 1;
    for(int i=1; i<intervals.length; i++){
        int head = queue.peek();
        if(intervals[i].start>=head){
            queue.poll();
        }else{
            count++;
        }
        queue.offer(intervals[i].end);
    }

    return count;
}
```

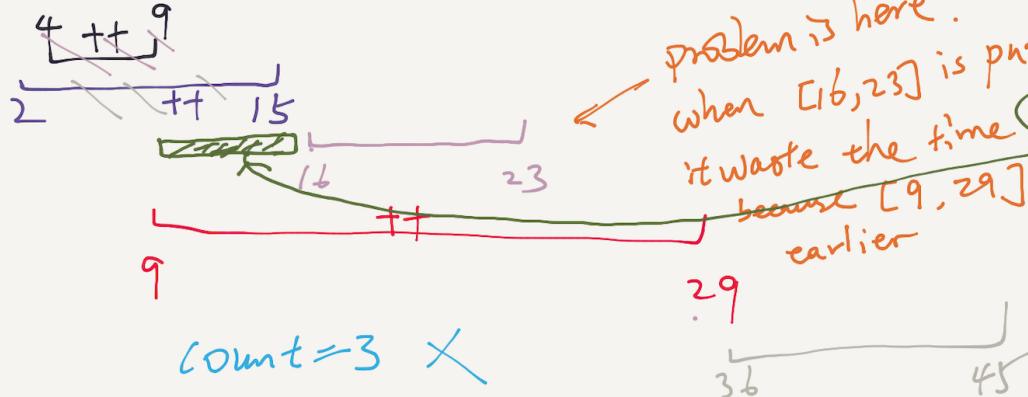
---

There was a discussion in the comment about why a regular queue is not good enough. I draw an example below to show why sorting based on start time and using a priority queue is necessary.

$[2, 15]$   $[36, 45]$   $[9, 29]$   $[16, 23]$   $[4, 9]$

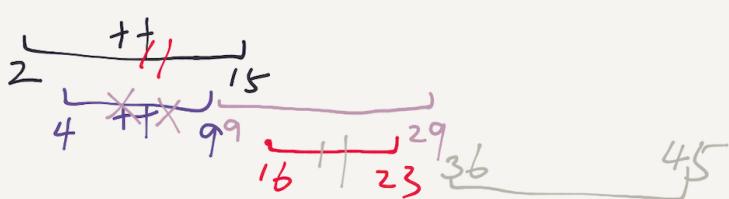
Sort end time

&  
Queue



Sort start time

Heap



Count = 2 ✓

# 66 Meeting Rooms

Given an array of meeting time intervals consisting of start and end times  $[s_1, e_1], [s_2, e_2], \dots$ , determine if a person could attend all meetings.

For example, Given  $[0, 30], [5, 10], [15, 20]$ , return false.

## 66.1 Java Solution

If a person can attend all meetings, there must not be any overlaps between any meetings. After sorting the intervals, we can compare the current end and next start.

---

```
public boolean canAttendMeetings(Interval[] intervals) {
    Arrays.sort(intervals, new Comparator<Interval>(){
        public int compare(Interval a, Interval b){
            return a.start-b.start;
        }
    });

    for(int i=0; i<intervals.length-1; i++){
        if(intervals[i].end>intervals[i+1].start){
            return false;
        }
    }

    return true;
}
```

---

# 67 Range Addition

Assume you have an array of length n initialized with all 0's and are given k update operations.

Each operation is represented as a triplet: [startIndex, endIndex, inc] which increments each element of subarray A[startIndex ... endIndex] (startIndex and endIndex inclusive) with inc.

Return the modified array after all k operations were executed.

For example, Input: length = 5, updates = [[1,3,2],[2,4,3],[0,2,-2]] Output: [-2,0,3,5,3]

## 67.1 Java Solution 1 - heap

```
public int[] getModifiedArray(int length, int[][][] updates) {
    int result[] = new int[length];
    if(updates==null || updates.length==0)
        return result;

    //sort updates by starting index
    Arrays.sort(updates, new Comparator<int[]>(){
        public int compare(int[] a, int [] b){
            return a[0]-b[0];
        }
    });

    ArrayList<int[]> list = new ArrayList<int[]>();

    //create a heap sorted by ending index
    PriorityQueue<Integer> queue = new PriorityQueue<Integer>(new Comparator<Integer>(){
        public int compare(Integer a, Integer b){
            return updates[a][1]-updates[b][1];
        }
    });

    int sum=0;
    int j=0;
    for(int i=0; i<length; i++){
        //subtract value from sum when ending index is reached
        while(!queue.isEmpty() && updates[queue.peek()][1] < i){
            int top = queue.poll();
            sum -= updates[top][2];
        }

        //add value to sum when starting index is reached
        while(j<updates.length && updates[j][0] <= i){
            sum = sum+updates[j][2];
            queue.offer(j);
            j++;
        }

        result[i]=sum;
    }
}
```

---

```

    return result;
}

```

---

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

## 67.2 Java Solution 2

We can track each range's start and end when iterating over the ranges. And in the final result array, adjust the values on the change points. The following shows an example:

	0	1	2	3	4
initially	0	0	0	0	0
$[0, 2, 2]$	2			-2	
$[1, 3, 3]$		3			-3
	↓	↓	↓	↓	↓
	2	$2+3=5$	$5+0=5$	$5-2=3$	$3-3=0$

---

```

public int[] getModifiedArray(int length, int[][][] updates) {
    int[] result = new int[length];
    if(updates==null||updates.length==0)
        return result;

    for(int i=0; i<updates.length; i++){
        result[updates[i][0]] += updates[i][2];
        if(updates[i][1]<length-1){
            result[updates[i][1]+1] -=updates[i][2];
        }
    }

    int v=0;
    for(int i=0; i<length; i++){
        v += result[i];
        result[i]=v;
    }

    return result;
}

```

---

Time complexity is  $O(n)$ .

## 68 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 complexity is  $O(n \log(k))$ , where  $n$  is the total number of elements and  $k$  is the number of arrays.

It takes  $O(\log(k))$  to insert an element to the heap and it takes  $O(\log(k))$  to delete the minimum element.

---

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

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

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

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

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

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

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

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

        return result;
    }

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

---

```
int[] arr1 = { 1, 3, 5, 7 };
int[] arr2 = { 2, 4, 6, 8 };
int[] arr3 = { 0, 9, 10, 11 };

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

---

# 69 Merge k Sorted Lists

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

## 69.1 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).

## 69.2 Java Solution

---

```
public ListNode mergeKLists(ListNode[] lists) {
    if(lists==null||lists.length==0)
        return null;

    PriorityQueue<ListNode> queue = new PriorityQueue<ListNode>(new Comparator<ListNode>(){
        public int compare(ListNode l1, ListNode l2){
            return l1.val - l2.val;
        }
    });

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

    for(ListNode list: lists){
        if(list!=null)
            queue.offer(list);
    }

    while(!queue.isEmpty()){
        ListNode n = queue.poll();
        p.next = n;
        p=p.next;

        if(n.next!=null)
            queue.offer(n.next);
    }

    return head.next;
}
```

---

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

In addition, from Java 8 comparator definition can be simplified as the following:

---

```
Comparator<ListNode> comp = Comparator.comparing(ListNode::getVal);
PriorityQueue<ListNode> queue = new PriorityQueue<*>(comp);
```

---

# 70 Rearrange String k Distance Apart

Given a non-empty string str and an integer k, rearrange the string such that the same characters are at least distance k from each other.

All input strings are given in lowercase letters. If it is not possible to rearrange the string, return an empty string "".

Example:

---

```
str = "aabbcc", k = 3
```

```
Result: "abcabc"
```

---

```
The same letters are at least distance 3 from each other.
```

## 70.1 Java Solution

---

```
public String rearrangeString(String str, int k) {
    if(k==0)
        return str;

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

    //sort the chars by frequency
    PriorityQueue<Character> queue = new PriorityQueue<Character>(new Comparator<Character>(){
        public int compare(Character c1, Character c2){
            if(map.get(c2).intValue()!=map.get(c1).intValue()){
                return map.get(c2)-map.get(c1);
            }else{
                return c1.compareTo(c2);
            }
        }
    });

    for(char c: map.keySet())
        queue.offer(c);

    StringBuilder sb = new StringBuilder();
    int len = str.length();
    int index = 0;
    while(len>0){
        if(queue.size()>0){
            char c = queue.poll();
            if(map.get(c).intValue()>1){
                map.put(c, map.get(c)-1);
            }else{
                map.remove(c);
            }
            sb.append(c);
            len--;
        }else{
            break;
        }
    }
    return sb.toString();
}
```

```
while(!queue.isEmpty()){

    int cnt = Math.min(k, len);
    ArrayList<Character> temp = new ArrayList<Character>();

    for(int i=0; i<cnt; i++){
        if(queue.isEmpty())
            return "";
        char c = queue.poll();
        sb.append(String.valueOf(c));

        map.put(c, map.get(c)-1);

        if(map.get(c)>0){
            temp.add(c);
        }

        len--;
    }

    for(char c: temp)
        queue.offer(c);
}

return sb.toString();
}
```

# 71 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.

## 71.1 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;
}
```

---

## 72 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  $k$ .

### 72.1 Java Solution 1 - HashMap

---

```
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;
        }

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

    return false;
}
```

---

### 72.2 Java Solution 2 - HashSet

---

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

    int i=0;

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

    for(int j=0; j<nums.length; j++){
        if(!set.add(nums[j])){
            return true;
        }

        if(set.size()>=k+1){
            set.remove(nums[i++]);
        }
    }

    return false;
}
```

---

# 73 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  $k$ .

## 73.1 Java Solution 1 - Simple

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){ // or if(set.size()>=k+1)
            set.remove((long)nums[i-k]);
        }
    }

    return false;
}
```

## 73.2 Java Solution 2 - Deprecated

1 2 4 5

$$\begin{aligned} \text{floor}(3) &= 2 \\ \text{ceiling}(3) &= 4 \end{aligned}$$

---

```
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;
}
```

---

# 74 Max Sum of Rectangle No Larger Than K

Given a non-empty 2D matrix matrix and an integer k, find the max sum of a rectangle in the matrix such that its sum is no larger than k.

Example:

---

```
Given matrix = [
    [1, 0, 1],
    [0, -2, 3]
]
k = 2
```

---

The answer is 2. Because the sum of rectangle  $[[0, 1], [-2, 3]]$  is 2 and 2 is the max number no larger than k ( $k = 2$ ).

Note: The rectangle inside the matrix must have an area  $> 0$ . What if the number of rows is much larger than the number of columns?

## 74.1 Analysis

We can solve this problem by comparing each submatrix. This method is trivial and we need a better solution. The key to the optimal solution is using a tree set to calculate the maximum sum of subarray close to k.

## 74.2 Java Solution 1

---

```
public int maxSumSubmatrix(int[][] matrix, int k) {
    if(matrix==null||matrix.length==0||matrix[0].length==0)
        return 0;

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

    int result = Integer.MIN_VALUE;

    for(int c1=0; c1<n; c1++){
        int[] each = new int[m];
        for(int c2=c1; c2>=0; c2--){
            for(int r=0; r<m; r++){
                each[r]+=matrix[r][c2];
            }
            result = Math.max(result, getLargestSumCloseToK(each, k));
        }
    }

    return result;
}

public int getLargestSumCloseToK(int[] arr, int k){
```

---

```

int sum=0;
TreeSet<Integer> set = new TreeSet<Integer>();
int result=Integer.MIN_VALUE;
set.add(0);

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

    Integer ceiling = set.ceiling(sum-k);
    if(ceiling!=null){
        result = Math.max(result, sum-ceiling);
    }

    set.add(sum);
}

return result;
}

```

---

The time complexity is  $O(n^2m \log(m))$ . If  $m$  is greater than  $n$ , this solution is fine. However, if  $m$  is less than  $n$ , then this solution is not optimal. In this case, we should reverse the row and column, like Solution 2.

### 74.3 Java Solution 2

---

```

public int maxSumSubmatrix(int[][] matrix, int k) {
    if(matrix==null||matrix.length==0||matrix[0].length==0)
        return 0;

    int row=matrix.length;
    int col=matrix[0].length;

    int m = Math.max(row, col);
    int n = Math.min(row, col);
    boolean isRowLarger = false;
    if(row>col)
        isRowLarger=true;

    int result = Integer.MIN_VALUE;

    for(int c1=0; c1<n; c1++){
        int[] each = new int[m];
        for(int c2=c1; c2>=0; c2--){
            for(int r=0; r<m; r++){
                each[r]+=isRowLarger?matrix[r][c2]:matrix[c2][r];
            }

            result = Math.max(result, getLargestSumCloseToK(each, k));
        }
    }

    return result;
}

public int getLargestSumCloseToK(int[] arr, int k){

```

---

```
int sum=0;
TreeSet<Integer> set = new TreeSet<Integer>();
int result=Integer.MIN_VALUE;
set.add(0);

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

    Integer ceiling = set.ceiling(sum-k);
    if(ceiling!=null){
        result = Math.max(result, sum-ceiling);
    }

    set.add(sum);
}

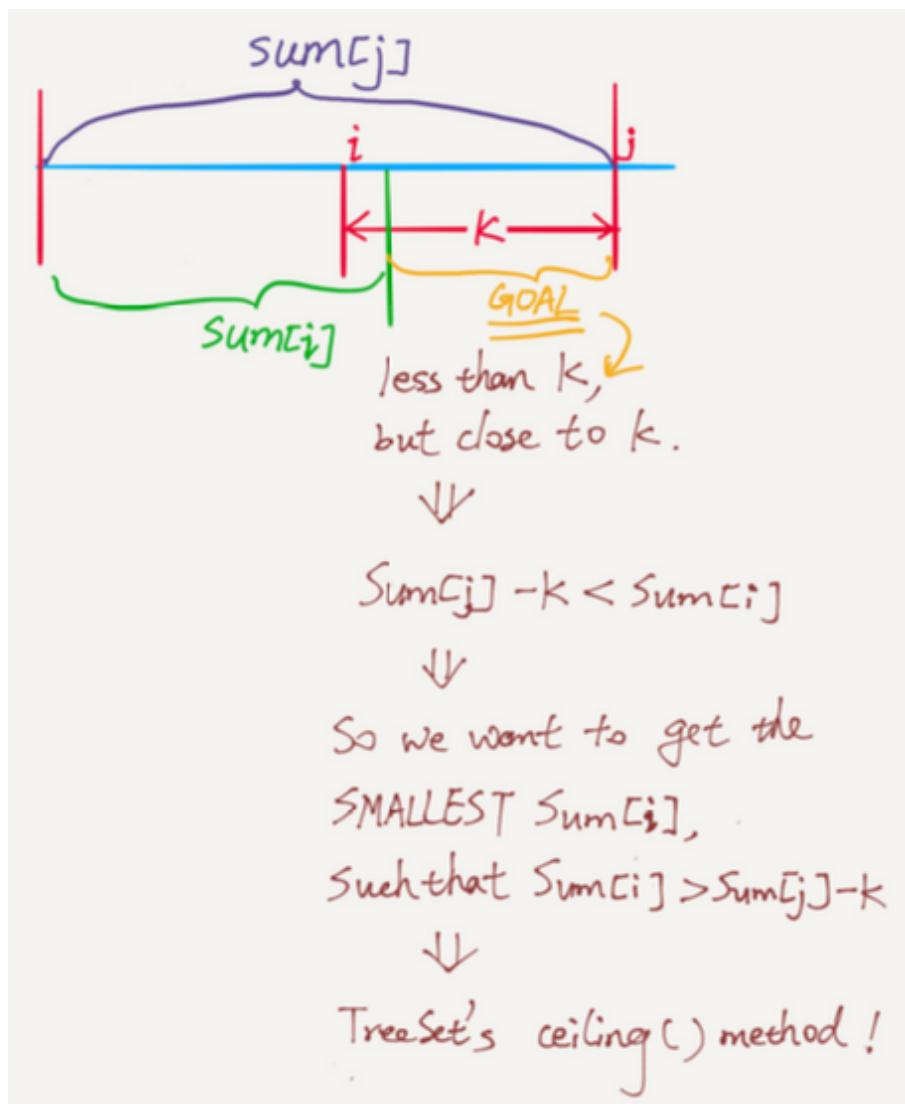
return result;
}
```

---

## 75 Maximum Sum of Subarray Close to K

Given an array, find the maximum sum of subarray close to k but not larger than k.

The solution to this problem is obvious when we draw the following diagram.



### 75.1 Java Solution

```
public int getLargestSumCloseToK(int[] arr, int k){  
    int sum=0;  
    TreeSet<Integer> set = new TreeSet<Integer>();  
    int result=Integer.MIN_VALUE;  
    set.add(0);
```

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

    Integer ceiling = set.ceiling(sum-k);
    if(ceiling!=null){
        result = Math.max(result, sum-ceiling);
    }

    set.add(sum);
}

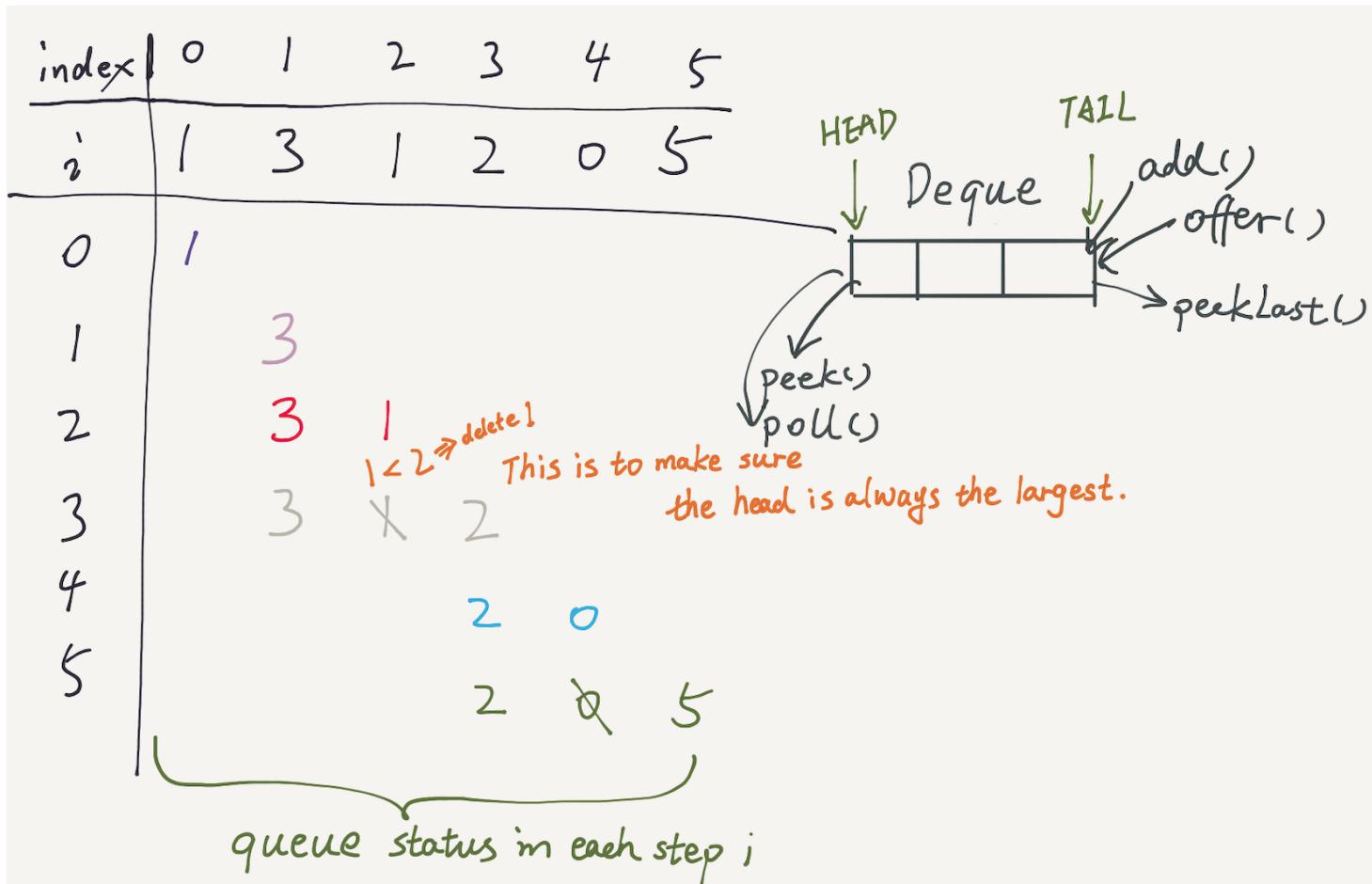
return result;
}
```

---

# 76 Sliding Window Maximum

Given an array `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

## 76.1 Java Solution



```
public int[] maxSlidingWindow(int[] nums, int k) {
    if(nums==null||nums.length==0)
        return new int[0];

    int[] result = new int[nums.length-k+1];

    LinkedList<Integer> deque = new LinkedList<Integer>();
    for(int i=0; i<nums.length; i++){

```

```
if(!deque.isEmpty()&&deque.peekFirst()==i-k)
    deque.poll();

while(!deque.isEmpty()&&nums[deque.peekLast()]<nums[i]){
    deque.removeLast();
}

deque.offer(i);

if(i+1>=k)
    result[i+1-k]=nums[deque.peek()];
}

return result;
}
```

---

# 77 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.

## 77.1 Java Solution

This problem is solved by using a queue.

---

```
public class MovingAverage {
    LinkedList<Integer> queue;
    int size;
    double avg;

    /** Initialize your data structure here. */
    public MovingAverage(int size) {
        this.queue = new LinkedList<Integer>();
        this.size = size;
    }

    public double next(int val) {
        if(queue.size()<this.size){
            queue.offer(val);
            int sum=0;
            for(int i: queue){
                sum+=i;
            }
            avg = (double)sum/queue.size();

            return avg;
        }else{
            int head = queue.poll();
            double minus = (double)head/this.size;
            queue.offer(val);
            double add = (double)val/this.size;
            avg = avg + add - minus;
            return avg;
        }
    }
}
```

---

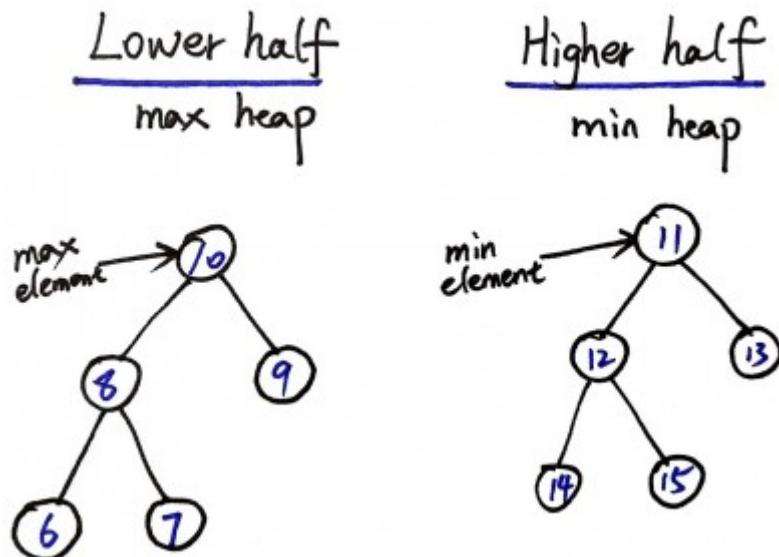
# 78 Find Median from Data Stream

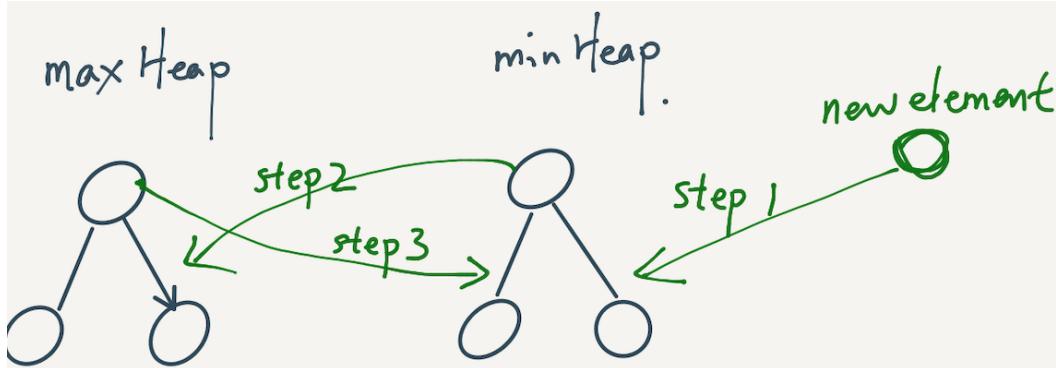
Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle values.

## 78.1 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.





Each element is added to minHeap first. (step 1)

Then the minimum element is popped out & offered to maxHeap (step 2)

This assures all elements in minHeap are greater than maxHeap.

Finally the two heaps need to be load balanced. (step 3)

```
class MedianFinder {
    PriorityQueue<Integer> minHeap = null;
    PriorityQueue<Integer> maxHeap = null;

    /** initialize your data structure here. */
    public MedianFinder() {
        minHeap = new PriorityQueue<>();
        maxHeap = new PriorityQueue<>(Comparator.reverseOrder());
    }

    public void addNum(int num) {
        minHeap.offer(num);
        maxHeap.offer(minHeap.poll());

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

    public double findMedian() {
        if(minHeap.size() > maxHeap.size()){
            return minHeap.peek();
        }else {
            return (minHeap.peek()+maxHeap.peek())/2.0;
        }
    }
}
```

# 79 Data Stream as Disjoint Intervals

Given a data stream input of non-negative integers  $a_1, a_2, \dots, a_n, \dots$ , summarize the numbers seen so far as a list of disjoint intervals.

For example, suppose the integers from the data stream are  $1, 3, 7, 2, 6, \dots$ , then the summary will be:

$[1, 1] [1, 1], [3, 3] [1, 1], [3, 3], [7, 7] [1, 3], [7, 7] [1, 3], [6, 7]$  Follow up: What if there are lots of merges and the number of disjoint intervals are small compared to the data stream's size?

## 79.1 Analysis

We can store the interval in an array and each time iterator over the array and merge the new value to an existing interval. This takes time  $O(n)$ . If there are a lot of merges, we want to do it in  $\log(n)$ .

We can solve this problem using a tree set. The `floor()` method returns the greatest element in this set less than or equal to the given element, or null if there is no such element. The `higher()` method returns the least element in this set strictly greater than the given element, or null if there is no such element. Note: we use `higher()` instead of `ceiling()` to exclude the given element.

## 79.2 Java Solution

```
public class SummaryRanges {  
  
    TreeSet<Interval> set;  
  
    /** Initialize your data structure here. */  
    public SummaryRanges() {  
        set = new TreeSet<Interval>(new Comparator<Interval>(){  
            public int compare(Interval a, Interval b){  
                return a.start-b.start;  
            }  
        });  
    }  
  
    public void addNum(int val) {  
        Interval t = new Interval(val, val);  
  
        Interval floor = set.floor(t);  
        if(floor!=null){  
            if(val<=floor.end){  
                return;  
            }else if(val == floor.end+1){  
                t.start = floor.start;  
                set.remove(floor);  
            }  
        }  
  
        Interval ceil = set.higher(t);  
        if(ceil!=null){  
            if(ceil.start==val+1){  
                t.end = ceil.end;  
            }  
        }  
    }  
}
```

```
        set.remove(ceil);
    }
}

set.add(t);
}

public List<Interval> getIntervals() {
    return new ArrayList(set);
//Arrays.asList(set.toArray(new Interval[0]));
}
}
```

---

# 80 Linked List Random Node

Given a singly linked list, return a random node's value from the linked list. Each node must have the same probability of being chosen.

Follow up: What if the linked list is extremely large and its length is unknown to you? Could you solve this efficiently without using extra space?

## 80.1 Java Solution

This problem is trivial, so I focus on the follow-up problem.

We want the probability of being chosen is  $1/\text{count}$ .

Given a list  $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow \dots \rightarrow n$ , the only time we can possibly select the third node is when the pointer points to 3. The probability of selecting the third node is  $1/3 * 3/4 * 4/5 * \dots * (n-1)/n = 1/n$ .

---

```
public class Solution {

    /** @param head The linked list's head. Note that the head is guaranteed to be not null, so it
     * contains at least one node. */
    Random r=null;
    ListNode h=null;
    public Solution(ListNode head) {
        r = new Random();
        h = head;
    }

    /** Returns a random node's value. */
    public int getRandom() {
        int count=1;
        ListNode p = h;
        int result = 0;
        while(p!=null){
            if(r.nextInt(count)==0)
                result= p.val;
            count++;
            p = p.next;
        }
        return result;
    }
}
```

---

## 81 Shuffle an Array

Shuffle a set of numbers without duplicates.

### 81.1 Java Solution

How we make sure each the probability of each element get shuffled is very similar to the [streaming random problem](#).

The algorithm is straightforward to understand, but the question is why it works. To have a working shuffle algorithm, every element in the array results in each position should be equal.

idx	0	1	2	3	4	5	6
value	0	1	2	3	4	5	6

The prob. of each number in each index should be equal to  $\frac{1}{n}$ .

For example,  $P(\text{number } 4 \text{ in index } 0) = \frac{1}{7}$

when  $i=0$ ,  $P(4 \text{ get selected}) = \frac{1}{7}$ , which is good. (if 4 is selected and put to idx 0 it does not change anymore)

when  $i=1$ ,  $P(4 \text{ get selected}) = \frac{6}{7} \times \frac{1}{6} = \frac{1}{7}$

⋮

$i=0$        $i=1$   
↑              ↑  
 $4 \text{ is not selected}$        $4 \text{ is selected}$ .

Same prob. applies to all other numbers in the array  
therefore the shuffling algorithm works.

```
class Solution {
    int[] original = null;
    int[] shuffle = null;
    Random rand = null;

    public Solution(int[] nums) {
        original = nums;
        shuffle = Arrays.copyOf(nums, nums.length);
```

```
rand = new Random();
}

/** Resets the array to its original configuration and return it. */
public int[] reset() {
    shuffle = Arrays.copyOf(original, original.length);
    return shuffle;
}

/** Returns a random shuffling of the array. */
public int[] shuffle() {
    for(int i=0; i<shuffle.length; i++){
        int x = rand.nextInt(shuffle.length-i);
        int idx = x+i;

        int tmp = shuffle[idx];
        shuffle[idx] = shuffle[i];
        shuffle[i] = tmp;
    }

    return shuffle;
}
}
```

---

# 82 Sort List

LeetCode - Sort List:

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

## 82.1 Analysis

If this problem does not have the constant space limitation, we can easily sort using a sorting method from Java SDK. With the constant space limitation, we need to do some pointer manipulation.

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

## 82.2 Java Solution

When I revisit this problem in 2018, I wrote it the following way which is more concise.

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

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

        //partition the list
        ListNode p1 = head;
        ListNode firstEnd = getFirstEnd(head);
        ListNode p2 = firstEnd.next;
        firstEnd.next = null;

        //sort each list
        p1 = sortList(p1);
        p2 = sortList(p2);

        //merge two lists
        return merge(p1, p2);
    }

    //get the list partition point
    private ListNode getFirstEnd(ListNode head){
        ListNode p1 = head;
        ListNode p2 = head;
```

```
while(p1!=null && p2!=null){
    if(p2.next==null||p2.next.next==null){
        return p1;
    }

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

return head;
}

//merge two list
private ListNode merge(ListNode n1, ListNode n2){
    ListNode head = new ListNode(0);
    ListNode p = head;
    ListNode p1 = n1;
    ListNode p2 = n2;
    while(p1!=null && p2!=null){
        if(p1.val<p2.val){
            p.next = p1;
            p1 = p1.next;
        }else{
            p.next = p2;
            p2 = p2.next;
        }

        p = p.next;
    }

    if(p1!=null){
        p.next = p1;
    }

    if(p2!=null){
        p.next = p2;
    }

    return head.next;
}
}
```

## 83 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
- Move smaller elements to the left and move bigger elements to the right of the pivot
- Recursively sort left part and right part

This post shows two versions of the Java implementation. The first one picks the rightmost element as the pivot and the second one picks the middle element as the pivot.

### 83.1 Version 1: Rightmost element as pivot

The following is the Java Implementation using rightmost element as the pivot.

```
public class QuickSort {

    public static void main(String[] args) {
        int[] arr = {4, 5, 1, 2, 3, 3};
        quickSort(arr, 0, arr.length-1);
        System.out.println(Arrays.toString(arr));
    }

    public static void quickSort(int[] arr, int start, int end){

        int partition = partition(arr, start, end);

        if(partition-1>start) {
            quickSort(arr, start, partition - 1);
        }
        if(partition+1<end) {
            quickSort(arr, partition + 1, end);
        }
    }

    public static int partition(int[] arr, int start, int end){
        int pivot = arr[end];

        for(int i=start; i<end; i++){
            if(arr[i]<pivot){
                int temp= arr[start];
                arr[start]=arr[i];
                arr[i]=temp;
                start++;
            }
        }

        int temp = arr[start];
```

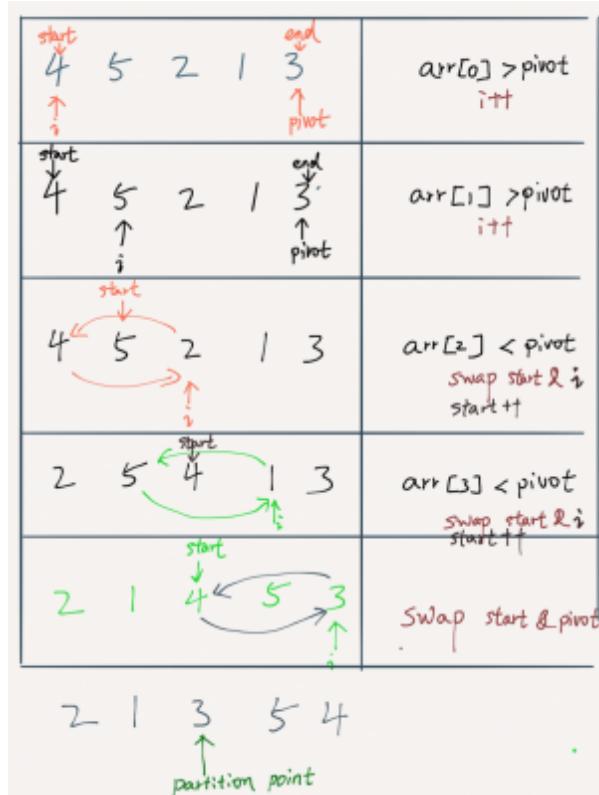
```

        arr[start] = pivot;
        arr[end] = temp;

        return start;
    }
}

```

You can use the example below to go through the code.



## 83.2 Version 2: Middle element as pivot

```

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

Here is a very good animation of quicksort.

# 84 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.

## 84.1 Java Solution 1 - Sorting

---

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

---

Time is  $O(n\log(n))$ . The problem of this solution is that sorting all elements is not necessary and is a overkill for getting just one element.

## 84.2 Java Solution 2 - Heap

We can use a min heap to solve this problem. The heap stores the top k largest elements. The top of the heap is the Kth Largest element and all other elements are greater than the heap top. 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();
}
```

---

Time complexity of  $n*\log(k)$  is an improvement to Solution 1. However, this solution requires  $O(k)$  space complexity and it is also maintained k-element heap.

## 84.3 Java Solution 3 - Quick Sort

This problem can also be solved by using a similar method like 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  $O(n^2)$ .

# 85 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.

## 85.1 Java Solution - Counting Sort

We can get the count of each element and project them to the original array.

---

```
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++;
        }
    }
}
```

---

# 86 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.

## 86.1 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.

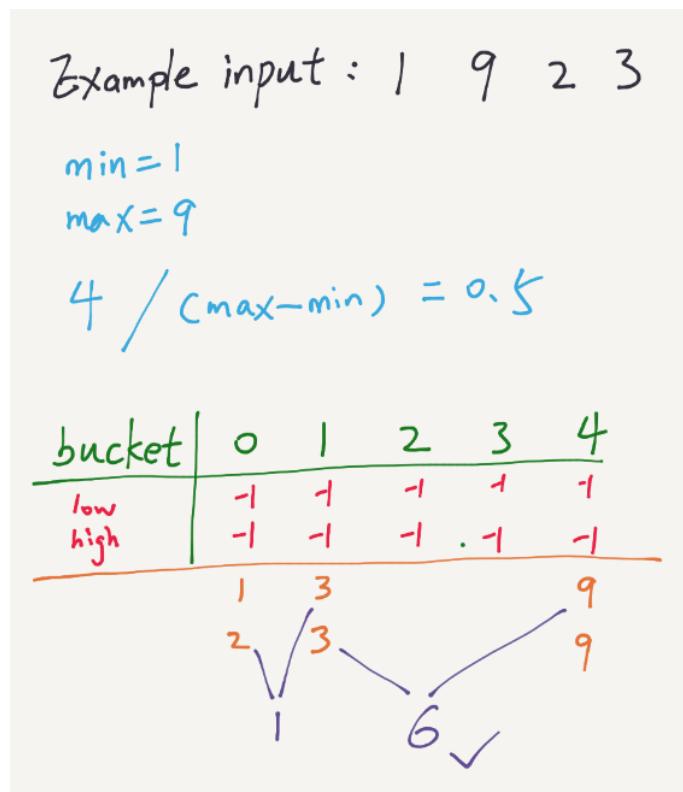
The key part is to get the interval:

---

```
From: interval * (num[i] - min) = 0 and interval * (max - num[i]) = n
interval = num.length / (max - min)
```

---

The following diagram shows an example.



## 86.2 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;
}

```

# 87 Group Anagrams

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

## 87.1 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.

## 87.2 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 = new char[26];
        for(int i=0; i<str.length(); i++){
            arr[str.charAt(i)-'a']++;
        }
        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);
        }
    }

    result.addAll(map.values());

    return result;
}
```

---

## 87.3 Time Complexity

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

## 88 Clone Graph

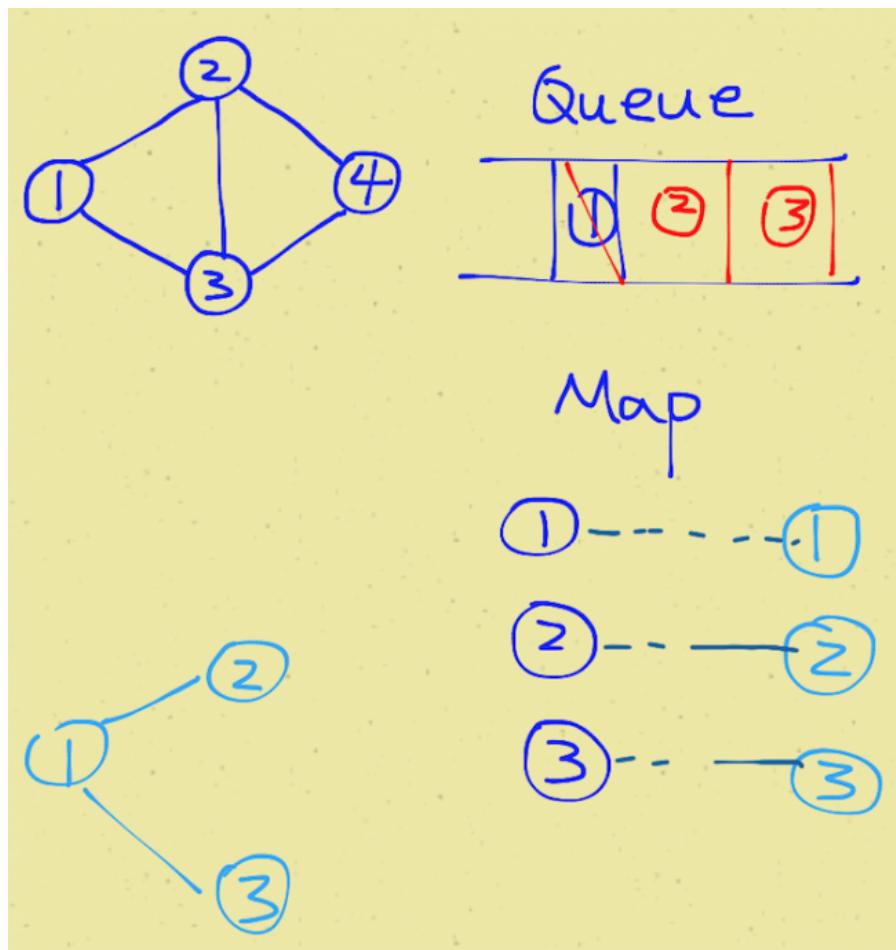
Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

### 88.1 Java Solution 1

In this solution,

- a queue is used to do breath first traversal,
- and 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;
    }
}

```

## 88.2 Java Solution 2

```

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
    if(node == null)
        return null;

    LinkedList<UndirectedGraphNode> queue = new LinkedList<UndirectedGraphNode>();

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

    queue.offer(node);
    while(!queue.isEmpty()){
        UndirectedGraphNode top = queue.poll();
        map.put(top, new UndirectedGraphNode(top.label));

        for(UndirectedGraphNode n: top.neighbors){

```

```
    if(!map.containsKey(n))
        queue.offer(n);
}
}

queue.offer(node);
HashSet<UndirectedGraphNode> set = new HashSet<UndirectedGraphNode>();
set.add(node);

while(!queue.isEmpty()){
    UndirectedGraphNode top = queue.poll();
    for(UndirectedGraphNode n: top.neighbors){
        if(!set.contains(n)){
            queue.offer(n);
            set.add(n);
        }
        map.get(top).neighbors.add(map.get(n));
    }
}

return map.get(node);
}
```

---

# 89 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.

## 89.1 Analysis

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

## 89.2 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);
        }
    }

    // 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;
}

```

---

### 89.3 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<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.

# 90 Course Schedule II

This is an extension of [Course Schedule](#). This time a valid sequence of courses is required as output.

## 90.1 Analysis

If we use the BFS solution of [Course Schedule](#), a valid sequence can easily be recorded.

## 90.2 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);
        }
    }

    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];
}
}
```

---

# 91 Minimum Height Trees

For a undirected graph with tree characteristics, we can choose any node as the root. The result graph is then a rooted tree. Among all possible rooted trees, those with minimum height are called minimum height trees (MHTs). Given such a graph, write a function to find all the MHTs and return a list of their root labels.

The graph contains n nodes which are labeled from 0 to n - 1. You will be given the number n and a list of undirected edges (each edge is a pair of labels).

You can assume that no duplicate edges will appear in edges. Since all edges are undirected, [0, 1] is the same as [1, 0] and thus will not appear together in edges.

Example 1:

---

```
Given n = 4, edges = [[1, 0], [1, 2], [1, 3]]
```

```
0
|
1
/
2 3
return [1]
```

---

## 91.1 Java Solution

---

```
public List<Integer> findMinHeightTrees(int n, int[][] edges) {
    List<Integer> result = new ArrayList<Integer>();
    if(n==0){
        return result;
    }
    if(n==1){
        result.add(0);
        return result;
    }

    ArrayList<HashSet<Integer>> graph = new ArrayList<HashSet<Integer>>();
    for(int i=0; i<n; i++){
        graph.add(new HashSet<Integer>());
    }

    for(int[] edge: edges){
        graph.get(edge[0]).add(edge[1]);
        graph.get(edge[1]).add(edge[0]);
    }

    LinkedList<Integer> leaves = new LinkedList<Integer>();
    for(int i=0; i<n; i++){
        if(graph.get(i).size()==1){
            leaves.offer(i);
        }
    }
```

```
if(leaves.size()==0){
    return result;
}

while(n>2){
    n = n-leaves.size();

    LinkedList<Integer> newLeaves = new LinkedList<Integer>();

    for(int l: leaves){
        int neighbor = graph.get(l).iterator().next();
        graph.get(neighbor).remove(l);
        if(graph.get(neighbor).size()==1){
            newLeaves.add(neighbor);
        }
    }

    leaves = newLeaves;
}

return leaves;
}
```

---

## 92 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.

### 92.1 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.

### 92.2 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);
    }
}
```

---

# 93 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.

## 93.1 Analysis

This problem can be converted to finding a cycle in a graph. It can be solved by using DFS (Recursion) or BFS (Queue).

## 93.2 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;
}
```

---

### 93.3 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;
}
```

---

## 94 Ugly Number

Write a program to check whether a given number is an ugly number. Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 6, 8 are ugly while 14 is not ugly since it includes another prime factor 7. Note that 1 is typically treated as an ugly number.

### 94.1 Java Solution

---

```
public boolean isUgly(int num) {  
    if(num==0) return false;  
    if(num==1) return true;  
  
    if(num%2==0){  
        num=num/2;  
        return isUgly(num);  
    }  
  
    if(num%3==0){  
        num=num/3;  
        return isUgly(num);  
    }  
  
    if(num%5==0){  
        num=num/5;  
        return isUgly(num);  
    }  
  
    return false;  
}
```

---

# 95 Ugly Number II

Write a program to find the n-th ugly number. Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10 ugly numbers. Note that 1 is typically treated as an ugly number.

## 95.1 Java Solution

---

```
public int nthUglyNumber(int n) {
    if(n<=0)
        return 0;

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

    int i=0;
    int j=0;
    int k=0;

    while(list.size()<n){
        int m2 = list.get(i)*2;
        int m3 = list.get(j)*3;
        int m5 = list.get(k)*5;

        int min = Math.min(m2, Math.min(m3, m5));
        list.add(min);

        if(min==m2)
            i++;

        if(min==m3)
            j++;

        if(min==m5)
            k++;
    }

    return list.get(list.size()-1);
}
```

---

# 96 Super Ugly Number

Write a program to find the nth super ugly number.

Super ugly numbers are positive numbers whose all prime factors are in the given prime list primes of size k. For example, [1, 2, 4, 7, 8, 13, 14, 16, 19, 26, 28, 32] is the sequence of the first 12 super ugly numbers given primes = [2, 7, 13, 19] of size 4.

Note: (1) 1 is a super ugly number for any given primes. (2) The given numbers in primes are in ascending order. (3) 0 < k <= 100, 0 < n <= 106, 0 < primes[i] < 1000.

## 96.1 Java Solution

Keep adding minimum values to results and updating the time value for the chosen prime number in each loop.

---

```
public int nthSuperUglyNumber(int n, int[] primes) {
    int[] times = new int[primes.length];
    int[] result = new int[n];
    result[0] = 1; // first is 1

    for (int i = 1; i < n; i++) {
        int min = Integer.MAX_VALUE;
        for (int j = 0; j < primes.length; j++) {
            min = Math.min(min, primes[j] * result[times[j]]);
        }

        result[i] = min;

        for (int j = 0; j < times.length; j++) {
            if (result[times[j]] * primes[j] == min) {
                times[j]++;
            }
        }
    }

    return result[n - 1];
}
```

---

# 97 Find K Pairs with Smallest Sums

You are given two integer arrays `nums1` and `nums2` sorted in ascending order and an integer `k`.

Define a pair  $(u,v)$  which consists of one element from the first array and one element from the second array.

Find the  $k$  pairs  $(u_1,v_1),(u_2,v_2) \dots (u_k,v_k)$  with the smallest sums.

Example:

---

Given `nums1 = [1,7,11]`, `nums2 = [2,4,6]`, `k = 3`

Return: `[1,2],[1,4],[1,6]`

The first 3 pairs are returned from the sequence:

`[1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]`

---

## 97.1 Java Solution

This problem is similar to [Super Ugly Number](#). The basic idea is using an array to track the index of the next element in the other array.

The best way to understand this solution is using an example such as  $\text{nums1}=[1,3,11]$  and  $\text{nums2}=[2,4,8]$ .

---

```
public List<int[]> kSmallestPairs(int[] nums1, int[] nums2, int k) {
    int total=nums1.length*nums2.length;
    if(total<k){
        k=total;
    }

    List<int[]> result = new ArrayList<int[]>();
    int[] idx = new int[nums1.length];//track each element's cursor in nums2
    while(k>0){
        int min=Integer.MAX_VALUE;
        int minIdx=-1;
        for(int i=0; i<nums1.length; i++){
            if(idx[i]<nums2.length && nums1[i]+nums2[idx[i]]<min){
                minIdx=i;
                min=nums1[i]+nums2[idx[i]];
            }
        }
        result.add(new int[]{nums1[minIdx],nums2[idx[minIdx]]});
        idx[minIdx]++;
        k--;
    }

    return result;
}
```

---

# 98 Rotate Array in Java

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?

## 98.1 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().

## 98.2 Solution 2 - Bubble Rotate

Can we do this in O(1) space?

This solution is like a bubble sort.

---

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

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

---

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

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
```

---

### 98.3 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--;
    }
}
```

```
    }  
}
```

---

# 99 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?

## 99.1 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--;
    }
}
```

---

# 100 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.

## 100.1 Java Solution 1 - Math

---

```
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;
}
```

---

## 100.2 Java Solution 2 - Bit

---

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

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

    return miss;
}
```

---

## 100.3 Java Solution 3 - Binary Search

---

```
public int missingNumber(int[] nums) {
    Arrays.sort(nums);
    int l=0, r=nums.length;
    while(l<r){
        int m = (l+r)/2;
        if(nums[m]>m){
            r=m;
        }else{
            l=m+1;
        }
    }

    return r;
}
```

---

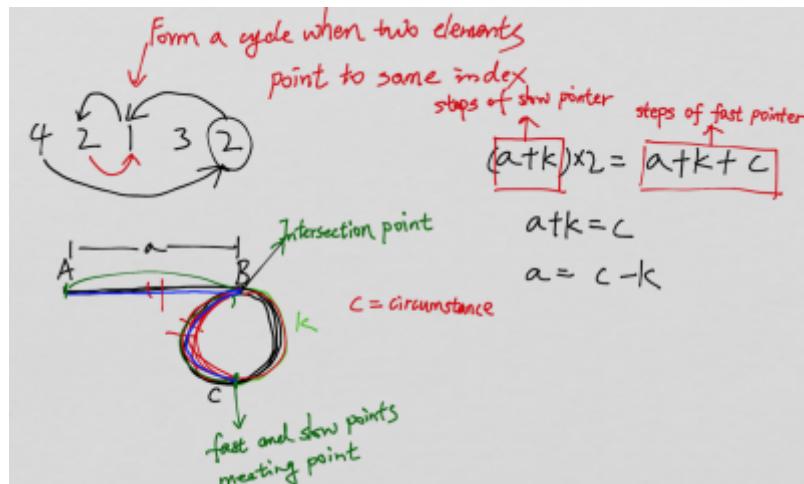
# 101 Find the Duplicate Number

Given an array containing  $n + 1$  integers where each integer is between 1 and  $n$  (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

Note: 1) You must not modify the array (assume the array is read only). 2) You must use only constant,  $O(1)$  extra space. 3) Your runtime complexity should be less than  $O(n^2)$ . 4) There is only one duplicate number in the array, but it could be repeated more than once.

## 101.1 Java Solution - Finding Cycle

The following shows how fast and slow pointers solution works. It basically contains 2 steps: 1) find the meeting point 2) start from the beginning and the meeting point respectively and find the intersection point.



---

```
public int findDuplicate(int[] nums) {
    int slow = 0;
    int fast = 0;

    do{
        slow = nums[slow];
        fast = nums[nums[fast]];
    } while(slow != fast);

    int find = 0;

    while(find != slow){
        slow = nums[slow];
        find = nums[find];
    }
    return find;
}
```

---

If we can assume there is only one duplicate number, it can be easily solved by using the sum of the array.

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

---

# 102 First Missing Positive

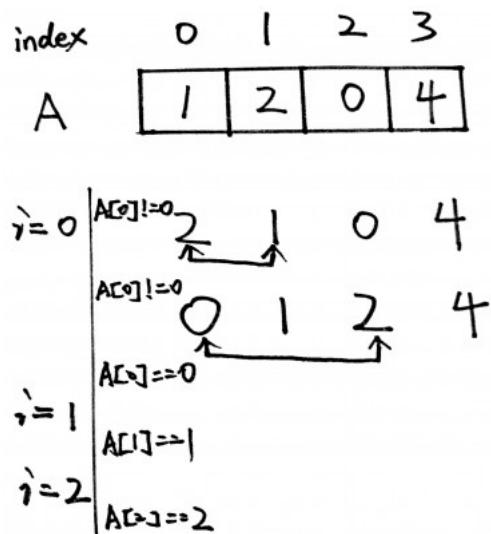
Given an unsorted integer array, find the first missing positive integer. For example, given [1,2,0] return 3 and [3,4,-1,1] return 2.

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

## 102.1 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  $i$ th 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[temp];
            A[temp] = temp;
        }
    }
}
```

---

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

---

## 102.2 Java Solution

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

---

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

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

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

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

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

    return n + 1;
}
```

---

# 103 Queue Reconstruction by Height

Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers ( $h$ ,  $k$ ), where  $h$  is the height of the person and  $k$  is the number of people in front of this person who have a height greater than or equal to  $h$ . Write an algorithm to reconstruct the queue.

Note: The number of people is less than 1,100.

Example

Input: [[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output: [[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

## 103.1 Analysis

The key to solve this problem is finding the start point of reconstruction. For this problem, we can start adding the largest element first. The basic idea is to keep adding the largest element each time, until all elements are in place.

## 103.2 Java Solution

---

```
public int[][] reconstructQueue(int[][][] people) {
    int[][] result = new int[people.length][];
    Arrays.sort(people, new Comparator<int[]>(){
        public int compare(int[] a1, int[] a2){
            if(a1[0]!=a2[0]){
                return a2[0]-a1[0];
            }else{
                return a1[1]-a2[1];
            }
        }
    });
    ArrayList<int[]> list = new ArrayList<int[]>();

    for(int i=0; i<people.length; i++){
        int[] arr = people[i];
        list.add(arr[1],arr);
    }

    for(int i=0; i<people.length; i++){
        result[i]=list.get(i);
    }

    return result;
}
```

---

Time complexity of this solution is  $O(n^2)$ .

# 104 Binary Watch

Given a non-negative integer n which represents the number of LEDs that are currently on, return all possible times the watch could represent.

Example:

Input: n = 1 Return: ["1:00", "2:00", "4:00", "8:00", "0:01", "0:02", "0:04", "0:08", "0:16", "0:32"]

## 104.1 Accepted Java Solution

---

```
public List<String> readBinaryWatch(int num) {
    List<String> result = new ArrayList<String>();

    for(int i=0; i<12; i++){
        for(int j=0; j<60; j++){
            int total = countDigits(i)+countDigits(j);
            if(total==num){
                String s="";
                s+=i+":";

                if(j<10){
                    s+="0"+j;
                }else{
                    s+=j;
                }

                result.add(s);
            }
        }
    }

    return result;
}

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

    while(num>0){
        if((num&1)==1){
            result++;
        }

        num>>=1;
    }

    return result;
}
```

---

Time complexity is constant ( $12^*60$ ).

## 104.2 Naive Solution

---

```

public class Solution {
    public List<String> readBinaryWatch(int num) {

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

        for(int i=0; i<=4; i++){
            int h=i;
            int m=num-i;

            ArrayList<ArrayList<Integer>> hSet = new ArrayList<ArrayList<Integer>>();
            subSet(h, 4, 1, new ArrayList<Integer>(), hSet);

            ArrayList<ArrayList<Integer>> mSet = new ArrayList<ArrayList<Integer>>();
            subSet(m, 6, 1, new ArrayList<Integer>(), mSet);

            ArrayList<String> hoursList = new ArrayList<String>();
            ArrayList<String> minsList = new ArrayList<String>();

            if(hSet.size()==0){
                hoursList.add("0");
            }else{
                hoursList.addAll(getTime(hSet, true));
            }

            if(mSet.size()==0){
                minsList.add("00");
            }else{
                minsList.addAll(getTime(mSet, false));
            }

            for(int x=0; x<hoursList.size(); x++){
                for(int y=0; y<minsList.size(); y++){
                    result.add(hoursList.get(x)+":"+minsList.get(y));
                }
            }
        }

        return result;
    }

    public ArrayList<String> getTime(ArrayList<ArrayList<Integer>> lists, boolean isHour){
        ArrayList<String> result = new ArrayList<String>();

        for(ArrayList<Integer> l : lists){
            int sum=0;
            for(int i: l){
                sum+= (1<<(i-1));
            }
            if(isHour && sum>=12)
                continue;

            if(!isHour&&sum>=60)

```

```
        continue;

    if(sum<10 && !isHour){
        result.add("0"+sum);
    }else{
        result.add("")+sum);
    }
}

return result;
}

public void subSet(int k, int m, int start, ArrayList<Integer> temp, ArrayList<ArrayList<Integer>>
result){
if(k==0){
    result.add(new ArrayList<Integer>(temp));
    return;
}
for(int i=start; i<=m; i++){
    temp.add(i);
    subSet(k-1, m, i+1, temp, result);
    temp.remove(temp.size()-1);
}
}
}
```

---

# 105 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.

## 105.1 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. The goal is to find the element in this sorted array by using binary search.

---

```
public class Solution {  
    public boolean searchMatrix(int[][] matrix, int target) {  
        if(matrix==null || matrix.length==0 || matrix[0].length==0)  
            return false;  
  
        int m = matrix.length;  
        int n = matrix[0].length;  
  
        int start = 0;  
        int end = m*n-1;  
  
        while(start<=end){  
            int mid=(start+end)/2;  
            int midX=mid/n;  
            int midY=mid%n;  
  
            if(matrix[midX][midY]==target)  
                return true;  
  
            if(matrix[midX][midY]<target){  
                start=mid+1;  
            }else{  
                end=mid-1;  
            }  
        }  
  
        return false;  
    }  
}
```

---

# 106 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.

## 106.1 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;
}

```

---

## 106.2 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;
}

```

---

# 107 Kth Smallest Element in a Sorted Matrix

Given a  $n \times n$  matrix where each of the rows and columns are sorted in ascending order, find the  $k$ th smallest element in the matrix.

Note that it is the  $k$ th smallest element in the sorted order, not the  $k$ th distinct element.

Example:

```
matrix = [
    [ 1, 5, 9],
    [10, 11, 13],
    [12, 13, 15]
],  
k = 8,  
return 13.
```

## 107.1 Java Solution

This problem is similar to [Search a 2D Matrix II](#). The start point of such a sorted matrix is left-bottom corner.

```
public int kthSmallest(int[][] matrix, int k) {
    int m=matrix.length;

    int lower = matrix[0][0];
    int upper = matrix[m-1][m-1];

    while(lower<upper){
        int mid = lower + ((upper-lower)>>1);
        int count = count(matrix, mid);
        if(count<k){
            lower=mid+1;
        }else{
            upper=mid;
        }
    }

    return upper;
}

private int count(int[][] matrix, int target){
    int m=matrix.length;
    int i=m-1;
    int j=0;
    int count = 0;

    while(i>=0&&j<m){
        if(matrix[i][j]<=target){
            count += i+1;
            j++;
        }else{
            i--;
        }
    }
}
```

```
        }
    }

    return count;
}
```

---

# 108 Design Snake Game

Design a Snake game that is played on a device with screen size = width x height. Play the game online if you are not familiar with the game.

The snake is initially positioned at the top left corner (0,0) with length = 1 unit.

You are given a list of food's positions in row-column order. When a snake eats the food, its length and the game's score both increase by 1.

Each food appears one by one on the screen. For example, the second food will not appear until the first food was eaten by the snake.

When a food does appear on the screen, it is guaranteed that it will not appear on a block occupied by the snake.

## 108.1 Java Solution

We can use a queue to track the snake positions.

```
public class SnakeGame {
    int[][][] food;
    int index;
    int row, col;
    int x, y;
    int len;
    LinkedList<int[]> queue;
    /** Initialize your data structure here.
     * @param width - screen width
     * @param height - screen height
     * @param food - A list of food positions
     * E.g food = [[1,1], [1,0]] means the first food is positioned at [1,1], the second is at [1,0]. */
    public SnakeGame(int width, int height, int[][][] food) {
        this.food=food;
        this.index=0;
        this.x=0;
        this.y=0;
        this.row=height;
        this.col=width;
        this.queue= new LinkedList<int[]>();
        this.queue.offer(new int[]{0, 0});
    }

    /** Moves the snake.
     * @param direction - 'U' = Up, 'L' = Left, 'R' = Right, 'D' = Down
     * @return The game's score after the move. Return -1 if game over.
     * Game over when snake crosses the screen boundary or bites its body. */
    public int move(String direction) {
        switch(direction){
            case "U":
                x--;
                break;
            case "L":
                y--;
                break;
```

```
        case "R":
            y++;
            break;
        case "D":
            x++;
            break;
    }

    if(!isValid(x,y)){
        return -1;
    }

    return process(x, y);
}

public boolean isValid(int x, int y){
    if(x<0 || x>=row || y<0 || y>=col)
        return false;

    return true;
}

public int process(int x, int y){

    if(index==food.length){
        queue.poll();

    }else if(food[index][0]==x && food[index][1]==y){
        len++;
        index++;
    }else{
        queue.poll();
    }

    for(int[] p: queue){
        if(p[0]==x&&p[1]==y)
            return -1;
    }

    queue.offer(new int[]{x,y});

    return len;
}
}
```

# 109 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.

## 109.1 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--;
                }
            }
        }
        result.add(count);
    }

    return result;
}

public int getRoot(int[] arr, int i){
    while(i!=arr[i]){
        i=arr[i];
    }
}
```

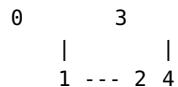
```
    return i;  
}
```

---

# 110 Number of Connected Components in an Undirected Graph

Given  $n$  nodes labeled from 0 to  $n - 1$  and a list of undirected edges (each edge is a pair of nodes), write a function to find the number of connected components in an undirected graph.

For example:



Given  $n = 5$  and edges =  $\{[0, 1], [1, 2], [3, 4]\}$ , return 2.

## 110.1 Java Solution - Union-find

This problem can be solved by using union-find beautifully. Initially, there are  $n$  nodes. The nodes that are involved in each edge are merged.

Example Input [0, 1] [2, 3] [3, 1]

Root array initialization

0	1	2	3
0	1	2	3

- [0, 1] 0's root = 0  
1's root = 1  
 $0 \neq 1$   
update 1's root to 0

0	1	2	3
0	1	2	3



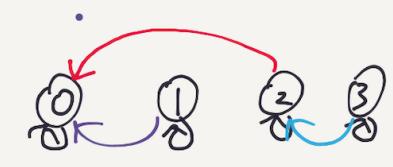
- [2, 3] 2's root = 2  
3's root = 3  
 $2 \neq 3$   
update 3's root to 2

0	0	2	3
0	0	2	3



- [3, 1] 1's root = 0  
2's root = 0  
update 2's root to 0

0	0	2	2
0	0	2	2



```
public int countComponents(int n, int[][][] edges) {
    int count = n;

    int[] root = new int[n];
    // initialize each node is an island
    for(int i=0; i<n; i++){
        root[i]=i;
    }

    for(int i=0; i<edges.length; i++){
        int x = edges[i][0];
        int y = edges[i][1];

        int xRoot = getRoot(root, x);
        int yRoot = getRoot(root, y);

        if(xRoot!=yRoot){
            count--;
            root[xRoot]=yRoot;
        }
    }
}
```

```
}

    return count;
}

public int getRoot(int[] arr, int i){
    while(arr[i]!=i){
        arr[i]= arr[arr[i]];
        i=arr[i];
    }
    return i;
}
```

There are k loops and each loop processing the root array costs  $\log(n)$ . Therefore, time complexity is  $O(k * \log(n))$ .

# 111 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

Example 1:

---

```
Input: nums =
[
  [9,9,4],
  [6,6,8],
  [2,1,1]
]
Output: 4
Explanation: The longest increasing path is [1, 2, 6, 9].
```

---

## 111.1 Java Solution 1 - Naive DFS

---

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

    int[] max = new int[1];
    for(int i=0; i<matrix.length; i++) {
        for(int j=0; j<matrix[0].length; j++){
            dfs(matrix, i, j, max, 1);
        }
    }

    return max[0];
}

public void dfs(int[][] matrix, int i, int j, int[] max, int len){
    max[0]=Math.max(max[0], len);

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

    int[] dx={-1, 0, 1, 0};
    int[] dy={0, 1, 0, -1};

    for(int k=0; k<4; k++){
        int x = i+dx[k];
        int y = j+dy[k];

        if(x>=0 && x<m && y>=0 && y<n && matrix[x][y]>matrix[i][j]){
            dfs(matrix, x, y, max, len+1);
        }
    }
}
```

```

    }
}
```

This naive DFS solution's time complexity is  $O(m^*n^*4^{(m+n)})$ .

## 111.2 Java Solution 2- Optimization with memorization

A common approach to improve DFS is through memorization.

```

public int longestIncreasingPath(int[][][] matrix) {
    if(matrix==null || matrix.length==0 || matrix[0].length==0){
        return 0;
    }

    int result = 1;

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

    for(int i=0; i<matrix.length; i++){
        for(int j=0; j<matrix[0].length; j++){
            result = Math.max(result, dfs(matrix, i, j, mem));
        }
    }

    return result;
}

private int dfs(int[][][] matrix, int i, int j, int[][] mem){
    if(mem[i][j]!=0){
        return mem[i][j];
    }

    int[] dx = {-1, 0, 1, 0};
    int[] dy = {0, 1, 0, -1};

    for(int k=0; k<4; k++){
        int x = i+dx[k];
        int y = j+dy[k];
        if(x>=0 && x<matrix.length
           && y>=0 && 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];
}
```

Because of the memorization matrix, the upper bound time complexity of the DFS is  $O(m^*n)$ . With the loop in the main method, the overall time complexity is  $O(m^2 * n^2)$

<table border="1"> <tr><td>9</td><td>9</td><td>4</td></tr> <tr><td>6</td><td>6</td><td>8</td></tr> <tr><td>2</td><td>1</td><td>1</td></tr> </table>	9	9	4	6	6	8	2	1	1	<table border="1"> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> </table>	1	0	0	0	0	0	0	0	0
9	9	4																	
6	6	8																	
2	1	1																	
1	0	0																	
0	0	0																	
0	0	0																	
<table border="1"> <tr><td>9</td><td>9</td><td>4</td></tr> <tr><td>6</td><td>6</td><td>8</td></tr> <tr><td>2</td><td>1</td><td>1</td></tr> </table>	9	9	4	6	6	8	2	1	1	<table border="1"> <tr><td>1</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> </table>	1	1	0	0	0	0	0	0	0
9	9	4																	
6	6	8																	
2	1	1																	
1	1	0																	
0	0	0																	
0	0	0																	
<table border="1"> <tr><td>9</td><td>9</td><td>4</td></tr> <tr><td>6</td><td>6</td><td>8</td></tr> <tr><td>2</td><td>1</td><td>1</td></tr> </table>	9	9	4	6	6	8	2	1	1	<table border="1"> <tr><td>1</td><td>1</td><td>2</td></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> </table>	1	1	2	0	0	1	0	0	0
9	9	4																	
6	6	8																	
2	1	1																	
1	1	2																	
0	0	1																	
0	0	0																	
<table border="1"> <tr><td>9</td><td>9</td><td>4</td></tr> <tr><td>6</td><td>6</td><td>8</td></tr> <tr><td>2</td><td>1</td><td>1</td></tr> </table>	9	9	4	6	6	8	2	1	1	<table border="1"> <tr><td>1</td><td>1</td><td>2</td></tr> <tr><td>2</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> </table>	1	1	2	2	0	1	0	0	0
9	9	4																	
6	6	8																	
2	1	1																	
1	1	2																	
2	0	1																	
0	0	0																	

## 112 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 = "ACB", ->returns false.

### 112.1 Java Solution

This problem can be solve by using a typical DFS algorithm.

```
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;  
    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)
```

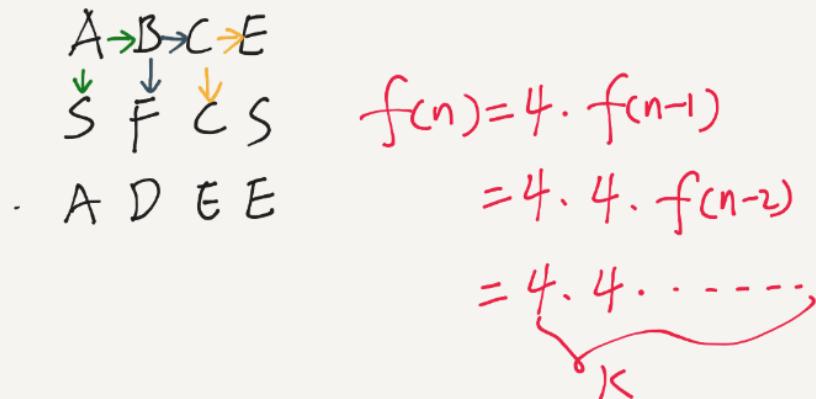
```

    ||dfs(board, word, i, j+1, k+1)){
        return true;
    }
    board[i][j]=temp;
}

return false;
}

```

Time of DFS :



*k* is the length of the target word.

DFS is called  $m \cdot n$

Therefore, overall time complexity is  $O(m \cdot n \cdot 4^k)$

Similarly, below is another way of writing this algorithm.

```

public boolean exist(char[][] board, String word) {
    for(int i=0; i<board.length; i++){
        for(int j=0; j<board[0].length; j++){
            if(dfs(board, word, i, j, 0)){
                return true;
            }
        }
    }

    return false;
}

public boolean dfs(char[][] board, String word, int i, int j, int k){
    if(board[i][j]!=word.charAt(k)){
        return false;
    }
}

```

```
if(k>=word.length()-1){  
    return true;  
}  
  
int[] di={-1,0,1,0};  
int[] dj={0,1,0,-1};  
  
char t = board[i][j];  
board[i][j]='#';  
  
for(int m=0; m<4; m++){  
    int pi=i+di[m];  
    int pj=j+dj[m];  
    if(pi>=0&&pi<board.length&&pj>=0&&pj<board[0].length&&board[pi][pj]==word.charAt(k+1)){  
        if(dfs(board,word,pi,pj,k+1)){  
            return true;  
        }  
    }  
}  
  
board[i][j]=t;  
  
return false;  
}
```

---

# 113 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"].

## 113.1 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;
}

```

---

## 113.2 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;
}
```

---

## 114 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

### 114.1 Java Solution 1 - DFS

The basic idea of the following solution is merging adjacent lands, and the merging should be done recursively. Each element is visited once only. So time is  $O(m*n)$ .

---

```
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){  
    int m=grid.length;  
    int n=grid[0].length;  
  
    if(i<0||i>=m||j<0||j>=n||grid[i][j]!='1')  
        return;  
  
    grid[i][j]='X';  
  
    merge(grid, i-1, j);  
    merge(grid, i+1, j);  
    merge(grid, i, j-1);
```

---

```
    merge(grid, i, j+1);
}
```

---

## 114.2 Java Solution 2 - Union-Find

Time is  $O(m \cdot n \cdot \log(k))$ .

```
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[] dx={-1, 1, 0, 0};
    int[] dy={0, 0, -1, 1};

    int[] root = new int[m*n];

    int count=0;
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(grid[i][j]=='1'){
                root[i*n+j] = i*n+j;
                count++;
            }
        }
    }

    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(grid[i][j]=='1'){
                for(int k=0; k<4; k++){
                    int x = i+dx[k];
                    int y = j+dy[k];

                    if(x>=0&&x<m&&y>=0&&y<n&&grid[x][y]=='1'){
                        int cRoot = getRoot(root, i*n+j);
                        int nRoot = getRoot(root, x*n+y);
                        if(nRoot!=cRoot){
                            root[cRoot]=nRoot; //update previous node's root to be current
                            count--;
                        }
                    }
                }
            }
        }
    }

    return count;
}

public int getRoot(int[] arr , int i){
    while(arr[i]!=i){
        i = arr[arr[i]];
    }
}
```

```
    }  
    return i;  
}
```

---

Check out [Number of Island II](#).

## 115 Find a Path in a Matrix

Given a 2d matrix, find a path from the top left corner to bottom right corner. Assume there exists at least one path, and you only need to find one valid path. You can move up, right, down, left at any position.

For example, given

---

```
[1, 0, 0, 0, 0]
[1, 0, 1, 1, 1]
[1, 1, 1, 0, 1]
[1, 0, 0, 0, 1]
[1, 0, 0, 0, 1]
```

---

A valid path is

---

```
[1, 0, 0, 0, 0]
[1, 0, 1, 1, 1]
[1, 1, 1, 0, 1]
[0, 0, 0, 0, 1]
[0, 0, 0, 0, 1]
```

---

### 115.1 Java Solution

---

```
public int[][] findPath(int[][] matrix){
    int m = matrix.length;

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

    ArrayList<int[]> temp = new ArrayList<int[]>();
    ArrayList<int[]> list = new ArrayList<int[]>();

    dfs(matrix, 0, 0, temp, list);

    for(int i=0; i<list.size(); i++){
        result[list.get(i)[0]][list.get(i)[1]]=1;
        //System.out.println(Arrays.toString(list.get(i)));
    }

    result[0][0]=1;

    return result;
}

public void dfs(int[][] matrix, int i, int j,
    ArrayList<int[]> temp, ArrayList<int[]> list){

    int m=matrix.length;

    if(i==m-1 && j==m-1){
```

```
list.clear();
list.addAll(temp);
return;
}

int[] dx = {-1, 0, 1, 0};
int[] dy = {0, 1, 0, -1};

for(int k=0; k<4; k++){
    int x = i+dx[k];
    int y = j+dy[k];

    if(x>=0&&y>=0&&x<=m-1&&y<=m-1 && matrix[x][y]==1){
        temp.add(new int[]{x,y});
        int prev = matrix[x][y];
        matrix[x][y]=0;

        dfs(matrix, x, y, temp, list);

        matrix[x][y]=prev;
        temp.remove(temp.size()-1);
    }
}
}
```

# 116 Sudoku Solver

Write a program to solve a Sudoku puzzle by filling the empty cells.

## 116.1 Java Solution

```
public void solveSudoku(char[][] board) {
    helper(board);
}

private boolean helper(char[][] board){
    for(int i=0; i<9; i++){
        for(int j=0; j<9; j++){
            if(board[i][j]!='.'){
                continue;
            }

            for(char k='1'; k<='9'; k++){
                board[i][j]=k;
                if(isValid(board, i, j) && helper(board)){
                    return true;
                }
                board[i][j]='.';
            }

            return false;
        }
    }

    return true; //return true if all cells are checked
}

private boolean isValid(char[][] board, int i, int j){
    HashSet<Character> set = new HashSet<>();

    for(int k=0; k<9; k++){
        if(set.contains(board[i][k])){
            return false;
        }
        if(board[i][k]!='.'){
            set.add(board[i][k]);
        }
    }

    set.clear();

    for(int k=0; k<9; k++){
        if(set.contains(board[k][j])){
            return false;
        }
        if(board[k][j]!='.'){
            set.add(board[k][j]);
        }
    }
}
```

```

    }
    if(board[k][j]!='.'){
        set.add(board[k][j]);
    }
}

set.clear();

int x=i/3 * 3;
int y=j/3 * 3;
for(int m=x; m<x+3; m++){
    for(int n=y; n<y+3; n++){
        if(set.contains(board[m][n])){
            return false;
        }
        if(board[m][n]!='.'){
            set.add(board[m][n]);
        }
    }
}
set.clear();

return true;
}

```

---

## 116.2 Improved Version

```

public void solveSudoku(char[][] board) {
    helper(board);
}

private boolean helper(char[][] board) {
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (board[i][j] != '.') {
                continue;
            }

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

    return true; //return true if all cells are checked
}

```

```
private boolean isValid(char[][] board, int row, int col, char c) {
    for (int i = 0; i < 9; i++) {
        if (board[i][col] != '.' && board[i][col] == c) {
            return false;
        }

        if (board[row][i] != '.' && board[row][i] == c) {
            return false;
        }

        if (board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] != '.'
            && board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c) {
            return false;
        }
    }
    return true;
}
```

The time complexity is  $O(9m)$  where  $m$  represents the number of blanks to be filled. No extra space is needed.

## 117 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	

### 117.1 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;
}
```

---

# 118 Walls and Gates

## 118.1 Java Solution 1 - DFS

```
public void wallsAndGates(int[][] rooms) {
    if(rooms==null || rooms.length==0 || rooms[0].length==0)
        return;

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

    boolean[][] visited = new boolean[m][n];

    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(rooms[i][j]==0){
                fill(rooms, i-1, j, 0, visited);
                fill(rooms, i, j+1, 0, visited);
                fill(rooms, i+1, j, 0, visited);
                fill(rooms, i, j-1, 0, visited);
                visited = new boolean[m][n];
            }
        }
    }
}

public void fill(int[][] rooms, int i, int j, int start, boolean[][] visited){
    int m=rooms.length;
    int n=rooms[0].length;

    if(i<0 || i>=m || j<0 || j>=n || rooms[i][j]<=0 || visited[i][j]){
        return;
    }

    rooms[i][j] = Math.min(rooms[i][j], start+1);
    visited[i][j]=true;

    fill(rooms, i-1, j, start+1, visited);
    fill(rooms, i, j+1, start+1, visited);
    fill(rooms, i+1, j, start+1, visited);
    fill(rooms, i, j-1, start+1, visited);

    visited[i][j]=false;
}
```

The DFS solution can be simplified as the following:

```
public void wallsAndGates(int[][] rooms) {
    if(rooms==null || rooms.length==0 || rooms[0].length==0)
        return;
```

```

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

for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
        if(rooms[i][j]==0){
            fill(rooms, i, j, 0);
        }
    }
}

public void fill(int[][] rooms, int i, int j, int distance){
    int m=rooms.length;
    int n=rooms[0].length;

    if(i<0||i>=m||j<0||j>=n||rooms[i][j]<distance){
        return;
    }

    rooms[i][j] = distance;

    fill(rooms, i-1, j, distance+1);
    fill(rooms, i, j+1, distance+1);
    fill(rooms, i+1, j, distance+1);
    fill(rooms, i, j-1, distance+1);
}

```

---

## 118.2 Java Solution 2 - BFS

```

public void wallsAndGates(int[][] rooms) {
    if(rooms==null || rooms.length==0|| rooms[0].length==0)
        return;

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

    LinkedList<Integer> queue = new LinkedList<Integer>();

    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(rooms[i][j]==0){
                queue.add(i*n+j);
            }
        }
    }

    while(!queue.isEmpty()){
        int head = queue.poll();
        int x=head/n;
        int y=head%n;

        if(x>0 && rooms[x-1][y]==Integer.MAX_VALUE){
            rooms[x-1][y]=rooms[x][y]+1;
            queue.add((x-1)*n+y);
        }
    }
}

```

```
}

if(x<m-1 && rooms[x+1][y]==Integer.MAX_VALUE){
    rooms[x+1][y]=rooms[x][y]+1;
    queue.add((x+1)*n+y);
}

if(y>0 && rooms[x][y-1]==Integer.MAX_VALUE){
    rooms[x][y-1]=rooms[x][y]+1;
    queue.add(x*n+y-1);
}

if(y<n-1 && rooms[x][y+1]==Integer.MAX_VALUE){
    rooms[x][y+1]=rooms[x][y]+1;
    queue.add(x*n+y+1);
}

}
```

---

# 119 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
```

---

## 119.1 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).

## 119.2 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 O's on top & bottom boarder
    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){
    board[i][j] = '#';

    int[] dx = {-1, 0, 1, 0};
    int[] dy = {0, 1, 0, -1};

    for(int k=0; k<4; k++){
        int x = i+dx[k];
        int y = j+dy[k];

        if(x>=0 && x<board.length
           && y>=0 && y<board[0].length
           && board[x][y]=='0'){
            merge(board, x, y);
        }
    }
}

```

---

### 119.3 Breath-first Search

We can also use a queue to do breath-first search for this problem.

```

public void solve(char[][] board) {
    if(board==null || board.length==0 || board[0].length==0)
        return;

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

    for(int j=0; j<n; j++){
        if(board[0][j]=='0'){
            bfs(board, 0, j);
        }
    }

    for(int j=0; j<n; j++){

```

```

        if(board[m-1][j]=='0'){
            bfs(board, m-1, j);
        }
    }

    for(int i=0; i<m; i++){
        if(board[i][0]=='0'){
            bfs(board, i, 0);
        }
    }

    for(int i=0; i<m; i++){
        if(board[i][n-1]=='0'){
            bfs(board, i, n-1);
        }
    }

    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(board[i][j]=='0'){
                board[i][j]='X';
            }
            if(board[i][j]=='1'){
                board[i][j]='0';
            }
        }
    }
}

public void bfs(char[][] board, int o, int p){
    int m=board.length;
    int n=board[0].length;

    int index = o*n+p;
    LinkedList<Integer> queue = new LinkedList<Integer>();
    queue.offer(index);
    board[o][p]='1';

    while(!queue.isEmpty()){
        int top = queue.poll();
        int i=top/n;
        int j=top%n;

        if(i-1>=0 && board[i-1][j]=='0'){
            board[i-1][j]='1';
            queue.offer((i-1)*n+j);
        }
        if(i+1<m && board[i+1][j]=='0'){
            board[i+1][j]='1';
            queue.offer((i+1)*n+j);
        }
        if(j-1>=0 && board[i][j-1]=='0'){
            board[i][j-1]='1';
            queue.offer(i*n+j-1);
        }
        if(j+1<n && board[i][j+1]=='0'){
            board[i][j+1]='1';
            queue.offer(i*n+j+1);
        }
    }
}

```

```
    }  
}  
}
```

---

# 120 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.

## 120.1 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

## 120.2 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;
}

}
```

# 121 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].

## 121.1 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;
}

```

---

Similarly, we can write the solution this way:

```

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

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

    int left=0;
    int right=n-1;
    int top = 0;
    int bottom = m-1;

    while(result.size()<m*n){
        for(int j=left; j<=right; j++){
            result.add(matrix[top][j]);
        }
        top++;

        for(int i=top; i<=bottom; i++){
            result.add(matrix[i][right]);
        }
        right--;

        //prevent duplicate row
        if(bottom<top)
            break;

        for(int j=right; j>=left; j--){
            result.add(matrix[bottom][j]);
        }
    }
}

```

```

    }

    bottom--;

    // prevent duplicate column
    if(right<left)
        break;

    for(int i=bottom; i>=top; i--){
        result.add(matrix[i][left]);
    }
    left++;
}

return result;
}

```

---

## 121.2 Java Solution 2

We can also recursively solve this problem. The solution's performance is not better than 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;
}
}
```

---

## 122 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]  
]
```

### 122.1 Java Solution 1

```
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++;  
            result[x][y]=i;  
            y--;  
        }  
        y++;  
        x--;  
        step++;  
  
        while(x>=0+step){
```

```

        i++;
        result[x][y]=i;
        x--;
    }
    x++;
    y++;
}
}

return result;
}

```

---

## 122.2 Java Solution 2

```

public int[][] generateMatrix(int n) {
    int[][] result = new int[n][n];

    int k=1;
    int top=0;
    int bottom=n-1;
    int left=0;
    int right=n-1;

    while(k<=n*n){
        for(int i=left; i<=right; i++){
            result[top][i]=k;
            k++;
        }
        top++;
        for(int i=top; i<=bottom; i++){
            result[i][right]=k;
            k++;
        }
        right--;
        for(int i=right; i>=left; i--){
            result[bottom][i]=k;
            k++;
        }
        bottom--;
        for(int i=bottom; i>=top; i--){
            result[i][left] = k;
            k++;
        }
        left++;
    }

    return result;
}

```

---

# 123 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?

## 123.1 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;
        }
    }
}
```

---

## 124 Range Sum Query 2D Immutable

Given a 2D matrix  $\text{matrix}$ , find the sum of the elements inside the rectangle defined by its upper left corner ( $\text{row1}, \text{col1}$ ) and lower right corner ( $\text{row2}, \text{col2}$ ).

### 124.1 Analysis

Since the assumption is that there are many calls to `sumRegion` method, we should use some extra space to store the intermediate results.

The solution is similar to other sum related problems such as . The basic idea is demonstrated in the following example:

—Immutable.png

./figures/Range-Sum-Query-2D-\T1\textendash -Immutable.png

Here we define an array sum[][] which stores the sum value from (0,0) to the current cell.

## 124.2 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;
}
}

```

# 125 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.

## 125.1 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>();
                    bfs(grid, i, j, i, j, 0, 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 bfs(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);
}
}

```

# 126 Best Meeting Point

A group of two or more people wants to meet and minimize the total travel distance. You are given a 2D grid of values 0 or 1, where each 1 marks the home of someone in the group. The distance is calculated using Manhattan Distance, where  $\text{distance}(p_1, p_2) = |p_2.x - p_1.x| + |p_2.y - p_1.y|$ .

For example, given three people living at (0,0), (0,4), and (2,2):

---

```
1 - 0 - 0 - 0 - 1
|   |   |   |
0 - 0 - 0 - 0 - 0
|   |   |   |
0 - 0 - 1 - 0 - 0
```

---

The point (0,2) is an ideal meeting point, as the total travel distance of  $2+2+2=6$  is minimal. So return 6.

## 126.1 Java Solution

This problem is converted to find the median value on x-axis and y-axis.

---

```
public int minTotalDistance(int[][] grid) {
    int m=grid.length;
    int n=grid[0].length;

    ArrayList<Integer> cols = new ArrayList<Integer>();
    ArrayList<Integer> rows = new ArrayList<Integer>();
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(grid[i][j]==1){
                cols.add(j);
                rows.add(i);
            }
        }
    }

    int sum=0;

    for(Integer i: rows){
        sum += Math.abs(i - rows.get(rows.size()/2));
    }

    Collections.sort(cols);

    for(Integer i: cols){
        sum+= Math.abs(i-cols.get(cols.size()/2));
    }

    return sum;
}
```

---

## 127 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.

### 127.1 Java Solution 1

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) {
    if(board==null || board.length==0 || board[0].length==0)
        return;

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

    int[] x = {-1, -1, 0, 1, 1, 1, 0, -1};
    int[] y = {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 nx=i+x[k];
                int ny=j+y[k];
                if(nx>=0&&nx<m&&ny>=0&&ny<n&&(board[nx][ny]&1)==1){
                    count++;
                }
            }

            //<2 die
            if(count<2){
                board[i][j] &= 1;
            }

            //same state
            if(count==2||count==3){
                board[i][j] |= board[i][j]<<1;
            }

            //go live
            if(count==3){
                board[i][j] |=2;
            }

            //>3 die
        }
    }
}
```

```
    if(count>3){
        board[i][j] &=1;
    }
}

for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
        board[i][j] = board[i][j]>>1;

    }
}
```

---

# 128 TicTacToe

Design a Tic-tac-toe game that is played between two players on a  $n \times n$  grid.

## 128.1 Java Solution 1 - Naive

We can simply check the row, column and the diagonals and see if there is a winner.

---

```
public class TicTacToe {

    int[][] matrix;

    /** Initialize your data structure here. */
    public TicTacToe(int n) {
        matrix = new int[n][n];
    }

    /** Player {player} makes a move at ({row}, {col}).
     * @param row The row of the board.
     * @param col The column of the board.
     * @param player The player, can be either 1 or 2.
     * @return The current winning condition, can be either:
     *         0: No one wins.
     *         1: Player 1 wins.
     *         2: Player 2 wins. */
    public int move(int row, int col, int player) {
        matrix[row][col]=player;

        //check row
        boolean win=true;
        for(int i=0; i<matrix.length; i++){
            if(matrix[row][i]!=player){
                win=false;
                break;
            }
        }

        if(win) return player;

        //check column
        win=true;
        for(int i=0; i<matrix.length; i++){
            if(matrix[i][col]!=player){
                win=false;
                break;
            }
        }

        if(win) return player;

        //check back diagonal
    }
}
```

```

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

        if(win) return player;

        //check forward diagonal
        win=true;
        for(int i=0; i<matrix.length; i++){
            if(matrix[i][matrix.length-i-1]!=player){
                win=false;
                break;
            }
        }

        if(win) return player;

        return 0;
    }
}

```

---

## 128.2 Java Solution 2

```

public class TicTacToe {
    int[] rows;
    int[] cols;
    int dc1;
    int dc2;
    int n;
    /** Initialize your data structure here. */
    public TicTacToe(int n) {
        this.n=n;
        this.rows=new int[n];
        this.cols=new int[n];
    }

    /** Player {player} makes a move at ({row}, {col}).
     * @param row The row of the board.
     * @param col The column of the board.
     * @param player The player, can be either 1 or 2.
     * @return The current winning condition, can be either:
     *         0: No one wins.
     *         1: Player 1 wins.
     *         2: Player 2 wins. */
    public int move(int row, int col, int player) {
        int val = (player==1?1:-1);

        rows[row]+=val;
        cols[col]+=val;

        if(row==col){

```

```
    dc1+=val;
}
if(col==n-row-1){
    dc2+=val;
}

if(Math.abs(rows[row])==n
|| Math.abs(cols[col])==n
|| Math.abs(dc1)==n
|| Math.abs(dc2)==n){
    return player;
}

return 0;
}
```

---

# 129 Sparse Matrix Multiplication

Given two sparse matrices A and B, return the result of AB.

You may assume that A's column number is equal to B's row number.

## 129.1 Naive Method

We can implement  $\text{Sum}(A_{ik} * B_{kj}) \rightarrow C_{ij}$  as a naive solution.

---

```
public int[][] multiply(int[][] A, int[][] B) {
    //validity check

    int[][] C = new int[A.length][B[0].length];

    for(int i=0; i<C.length; i++){
        for(int j=0; j<C[0].length; j++){
            int sum=0;
            for(int k=0; k<A[0].length; k++){
                sum += A[i][k]*B[k][j];
            }
            C[i][j] = sum;
        }
    }

    return C;
}
```

---

Time complexity is  $O(n^3)$ .

## 129.2 Optimized Method

From the formula:  $\text{Sum}(A_{ik} * B_{kj}) \rightarrow C_{ij}$

We can see that when  $A_{ik}$  is 0, there is no need to compute  $B_{kj}$ . So we switch the inner two loops and add a 0-checking condition.

---

```
public int[][] multiply(int[][] A, int[][] B) {
    //validity check

    int[][] C = new int[A.length][B[0].length];

    for(int i=0; i<C.length; i++){
        for(int k=0; k<A[0].length; k++){
            if(A[i][k]!=0){
                for(int j=0; j<C[0].length; j++){
                    C[i][j] += A[i][k]*B[k][j];
                }
            }
        }
    }

    return C;
}
```

---

```
    return C;
}
```

Since the matrix is sparse, time complexity is  $O(n^2)$  which is much faster than  $O(n^3)$ .

# 130 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

## 130.1 Java Solution

```
/*
2 -> 4 -> 3
5 -> 6 -> 4
7   0   8
*/
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode fake = new ListNode(0);
    ListNode p = fake;

    ListNode p1 = l1;
    ListNode p2 = l2;

    int carry = 0;
    while(p1!=null || p2!=null){
        int sum = carry;
        if(p1!=null){
            sum += p1.val;
            p1 = p1.next;
        }

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

        if(sum>9){
            carry=1;
            sum = sum-10;
        }else{
            carry = 0;
        }

        ListNode l = new ListNode(sum);
        p.next = l;
        p = p.next;
    }

    //don't forget check the carry value at the end
    if(carry > 0){
        ListNode l = new ListNode(carry);
        p.next = l;
    }
}
```

```
    return fake.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.

# 131 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.

## 131.1 Java Solution Because the problem requires "in-place" operations, we can only change their pointers, not creating a new list. This problem can be solved by doing the following 3 steps:

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

The following code is a complete runnable class with testing.

---

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

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

public class ReorderList {

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

        printList(n1);

        reorderList(n1);

        printList(n1);
    }

    public static void reorderList(ListNode head) {
        if (head != null && head.next != null) {
```

```

ListNode slow = head;
ListNode fast = head;

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

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

// now should have two lists: head and fast

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

ListNode p1 = head;
ListNode p2 = second;

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

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

    p1 = temp1;
    p2 = temp2;
}
}

}

public static ListNode reverseOrder(ListNode head) {

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

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

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

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

return pre;
}

```

```

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

```

## 131.2 Takeaway Messages

The three steps can be used to solve other problems of linked list. The following diagrams illustrate how each of the steps works.

Reverse List:

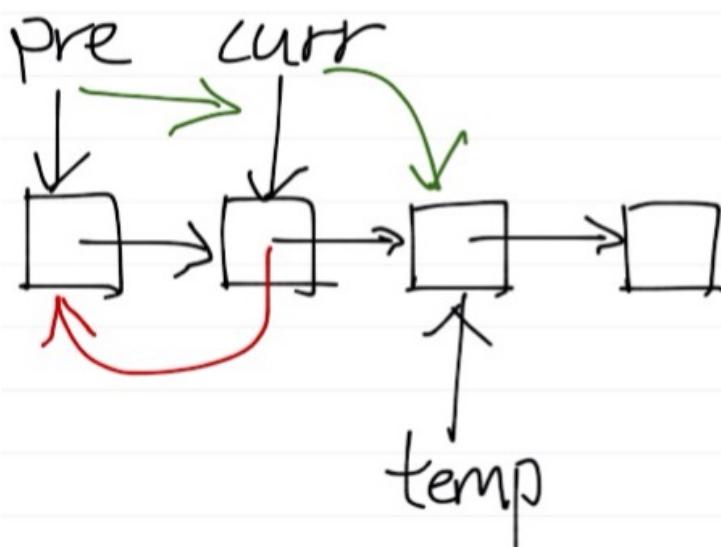
```

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:

```

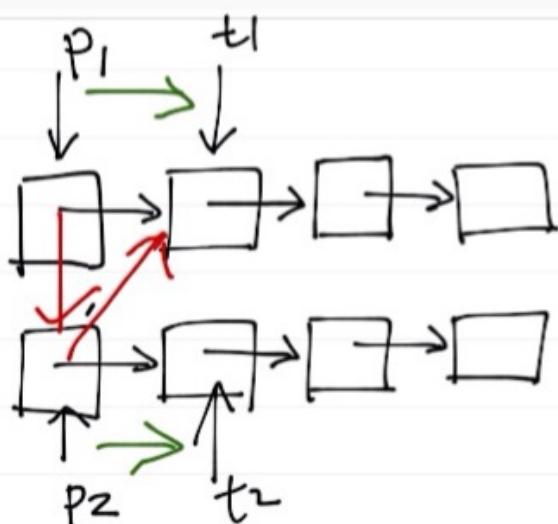
ListNode p1 = head;
ListNode p2 = second;

//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;
}

```



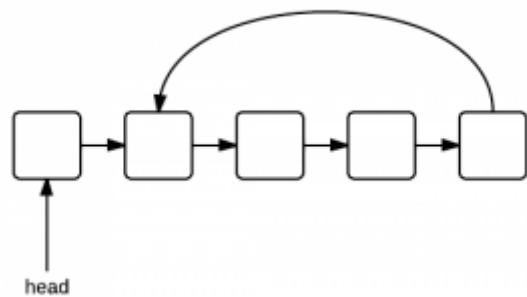
Note that pointers movements always starts with assigning the next node to a temporary variable t.

## 132 Linked List Cycle

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

### 132.1 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.



### 132.2 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;
    }
}
```

---

# 133 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.

## 133.1 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;
}
```

---

## 133.2 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;
}
```

---

# 134 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.

## 134.1 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.

## 134.2 Java Solution

---

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode head = new ListNode(0);
    ListNode p = head;

    while(l1!=null||l2!=null){
        if(l1!=null&&l2!=null){
            if(l1.val < l2.val){
                p.next = l1;
                l1=l1.next;
            }else{
                p.next=l2;
                l2=l2.next;
            }
            p = p.next;
        }else if(l1==null){
            p.next = l2;
            break;
        }else if(l2==null){
            p.next = l1;
            break;
        }
    }

    return head.next;
}
```

---

Or write this way:

---

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode head = new ListNode(0);
    ListNode p=head;

    ListNode p1=l1;
    ListNode p2=l2;
    while(p1!=null && p2!=null){
        if(p1.val < p2.val){
```

```
    p.next = p1;
    p1 = p1.next;
}else{
    p.next = p2;
    p2 = p2.next;
}
p=p.next;
}

if(p1!=null){
    p.next = p1;
}

if(p2!=null){
    p.next = p2;
}

return head.next;
}
```

---

# 135 Odd Even Linked List

## 135.1 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:

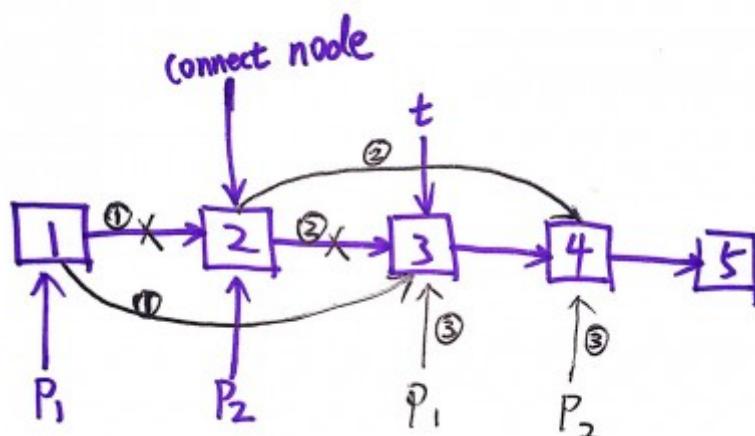
---

```
Given 1->2->3->4->5->NULL,
return 1->3->5->2->4->NULL.
```

---

## 135.2 Analysis

This problem can be solved by using two pointers. We iterate over the link and move the two pointers.



## 135.3 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 = t;
        p2.next = t.next;
        connectNode.next = p2;
        connectNode = p2;
        p1 = p1.next;
        p2 = p2.next;
    }
}
```

```
p1.next = p2.next;
p1 = p1.next;

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

p1.next = connectNode;

return result;
}
```

---

# 136 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.

---

## 136.1 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.

## 136.2 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;
    }
}
```

---

### 136.3 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;
    }
}
```

---

# 137 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`.

## 137.1 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;
}
```

---

# 138 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  $1 \rightarrow 2 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5$ .

## 138.1 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;
    }
}
```

---

# 139 Intersection of Two Linked Lists

## 139.1 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.

## 139.2 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;
        }

        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;
}
```

---

# 140 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

---

## 140.1 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;
}
```

---

# 141 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.

## 141.1 Java Solution 1

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;
}
```

---

## 141.2 Java Solution 2

Each time I do the same problem I often get the different solutions. Here is another way of writing this solution.

---

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

    //a fake head
    ListNode h = new ListNode(0);
    h.next = head;

    ListNode p1 = head;
    ListNode p2 = head.next;
```

```
ListNode pre = h;
while(p1!=null && p2!=null){
    pre.next = p2;

    ListNode t = p2.next;
    p2.next = p1;
    pre = p1;
    p1.next = t;

    p1 = p1.next;

    if(t!=null)
        p2 = t.next;
}

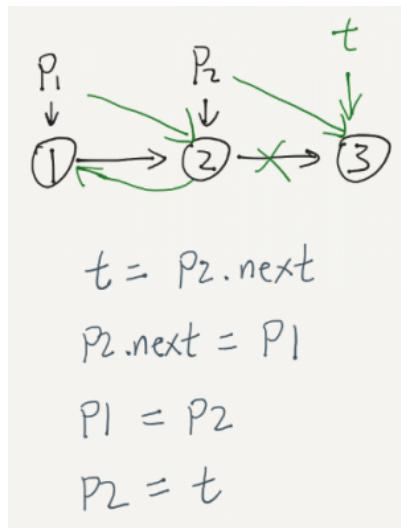
return h.next;
```

---

## 142 Reverse Linked List

Reverse a singly linked list.

### 142.1 Java Solution 1 - Iterative



---

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

    ListNode p1 = head;
    ListNode p2 = p1.next;

    head.next = null;
    while(p1!=null && p2!=null){
        ListNode t = p2.next;
        p2.next = p1;
        p1 = p2;
        p2 = t;
    }

    return p1;
}
```

---

### 142.2 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);
second.next = head;

return rest;
}
```

---

# 143 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.

## 143.1 Analysis

## 143.2 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;
    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;

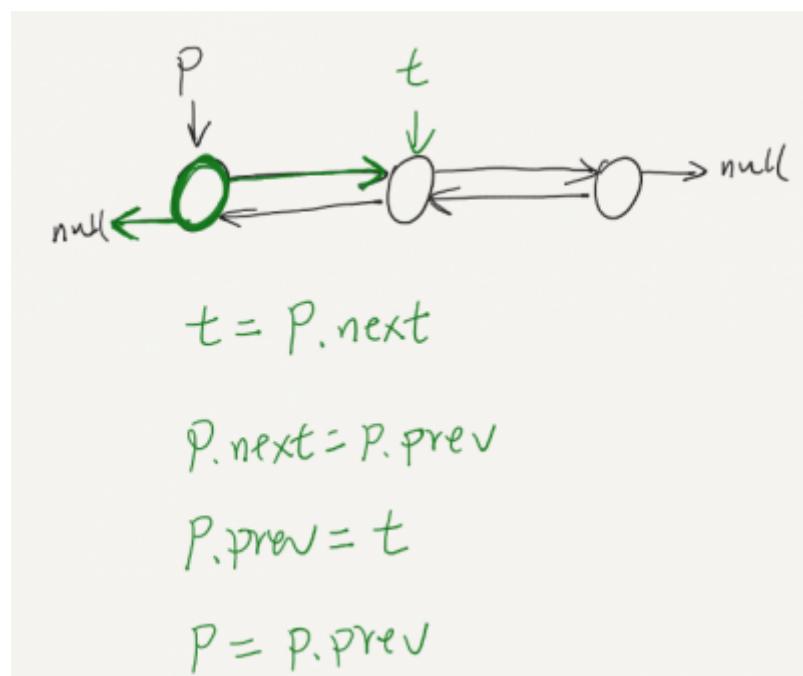
return head;
}
```

---

## 144 Reverse Double Linked List

Given a double linked list's head node, reverse the list and return the new head node.

### 144.1 Java Solution



```
/*
 * For your reference:
 *
 * DoublyLinkedListNode {
 *     int data;
 *     DoublyLinkedListNode next;
 *     DoublyLinkedListNode prev;
 * }
 *
 */
static DoublyLinkedListNode reverse(DoublyLinkedListNode head) {
    DoublyLinkedListNode p = head;
    DoublyLinkedListNode newHead = head;

    while(p!=null){
        DoublyLinkedListNode t = p.next;
        p.next = p.prev;
        p.prev = t;
```

```
    newHead= p;
    p = t;
}

return newHead;
}
```

---

# 145 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.

## 145.1 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;
}
```

## 145.2 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;
}
```

---

# 146 Palindrome Linked List

Given a singly linked list, determine if it is a palindrome.

## 146.1 Java Solution 1 - Create a new reversed list

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;
}
```

---

## 146.2 Java Solution 2 - Break and reverse second half

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;
}

```

---

### 146.3 Java Solution 3 - Recursive

```

public class Solution {
    ListNode left;

    public boolean isPalindrome(ListNode head) {
        left = head;

        boolean result = helper(head);
        return result;
    }

    public boolean helper(ListNode right){

        //stop recursion
        if (right == null)
            return true;

```

```
//if sub-list is not palindrome, return false
boolean x = helper(right.next);
if (!x)
    return false;

//current left and right
boolean y = (left.val == right.val);

//move left to next
left = left.next;

return y;
}
```

---

Time is  $O(n)$  and space is  $O(n)$ .

## 147 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.

### 147.1 Java Solution

---

```
public void deleteNode(ListNode node) {  
    node.val = node.next.val;  
    node.next = node.next.next;  
}
```

---

## 148 Reverse Nodes in kGroup

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list. If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is. You may not alter the values in the nodes, only nodes itself may be changed.

For example,

---

```
Given this linked list: 1->2->3->4->5
For k = 2, you should return: 2->1->4->3->5
For k = 3, you should return: 3->2->1->4->5
```

---

### 148.1 Java Solution

---

```
public ListNode reverseKGroup(ListNode head, int k) {
    if(head==null||k==1)
        return head;

    ListNode fake = new ListNode(0);
    fake.next = head;
    ListNode prev = fake;
    int i=0;

    ListNode p = head;
    while(p!=null){
        i++;
        if(i%k==0){
            prev = reverse(prev, p.next);
            p = prev.next;
        }else{
            p = p.next;
        }
    }

    return fake.next;
}
```

---

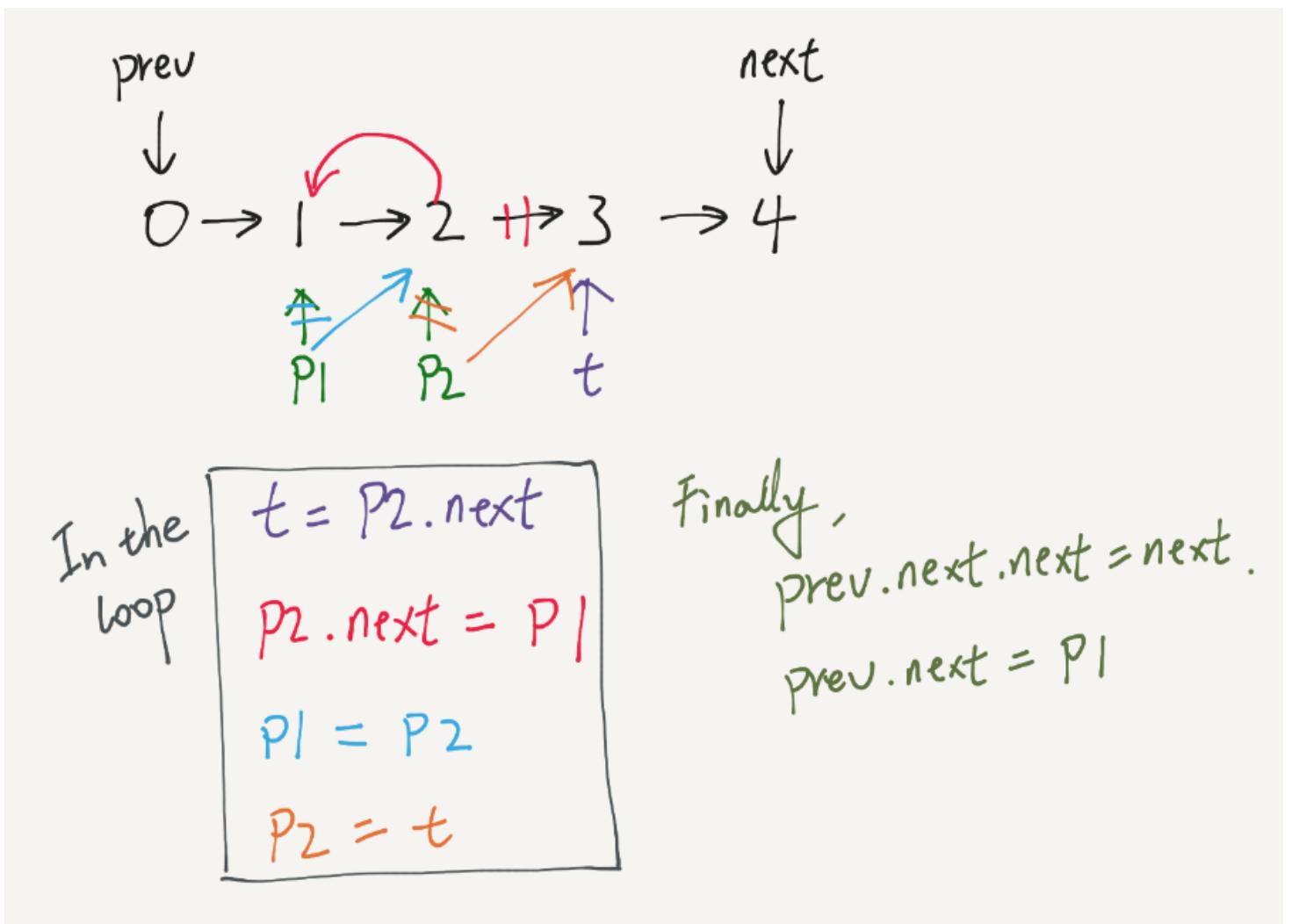
```
public ListNode reverse(ListNode prev, ListNode next){
    ListNode last = prev.next;
    ListNode curr = last.next;

    while(curr != next){
        last.next = curr.next;
        curr.next = prev.next;
        prev.next = curr;
        curr = last.next;
    }

    return last;
```

}

We can write the reverse method differently like the following. I personally it is more understandable.



```
private ListNode reverse(ListNode prev, ListNode next){
    ListNode p1 = prev.next;
    ListNode p2 = p1.next;

    while(p2 != next){
        ListNode t = p2.next;
        p2.next = p1;
        p1 = p2;
        p2 = t;
    }

    ListNode rNode = prev.next;
    prev.next.next = next;
    prev.next = p1;

    return rNode;
}
```

}

## 149 Plus One Linked List

Given a non-negative number represented as a singly linked list of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

Example:

---

Input:

1->2->3

Output:

1->2->4

---

### 149.1 Java Solution

```
public ListNode plusOne(ListNode head) {
    ListNode h2 = reverse(head);

    ListNode p=h2;

    while(p!=null){
        if(p.val+1<=9){
            p.val=p.val+1;
            break;
        }else{
            p.val=0;
            if(p.next==null){
                p.next = new ListNode(1);
                break;
            }
            p=p.next;
        }
    }

    return reverse(h2);
}

public ListNode reverse(ListNode head){
    if(head==null||head.next==null)
        return head;

    ListNode p1=head;
    ListNode p2=p1.next;
    while(p2!=null){
        ListNode t = p2.next;
        p2.next=p1;
        p1=p2;
        p2=t;
    }
}
```

```
head.next=null;  
    return p1;  
}
```

---

# 150 Binary Tree Preorder Traversal

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

## 150.1 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;
    }
}
```

---

# 151 Binary Tree Inorder Traversal

There are 3 solutions for solving this problem.

## 151.1 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

---

```
public List<Integer> inorderTraversal(TreeNode root) {  
    ArrayList<Integer> result = new ArrayList<>();  
    Stack<TreeNode> stack = new Stack<>();  
  
    TreeNode p = root;  
    while(p!=null){  
        stack.push(p);  
        p=p.left;  
    }  
  
    while(!stack.isEmpty()){  
        TreeNode t = stack.pop();  
        result.add(t.val);  
  
        t = t.right;  
        while(t!=null){  
            stack.push(t);  
            t = t.left;  
        }  
    }  
  
    return result;  
}
```

---

## 151.2 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);  
}  
}
```

---

### 151.3 Java Solution 3 - Simple

The following solution is simple, but it changes the tree structure, i.e., remove pointers to left and right children.

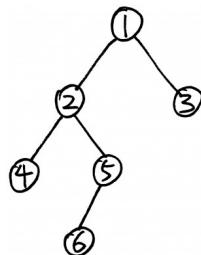
---

```
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;  
}
```

---

## 152 Binary Tree Postorder Traversal

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



### 152.1 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 left, 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;
}
}

```

## 152.2 Java Solution 2 - Simple!

This solution is simple, but note that the tree's structure gets changed since children are set to null.

```

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;
        }
    }
}
return res;
```

---

# 153 Binary Tree Level Order Traversal

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

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

---

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

---

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

## 153.1 Java Solution 1

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.

---

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

---

## 153.2 Java Solution 2

We can also improve Solution 1 by use a queue of integer to track the level instead of track another level of nodes.

---

```

public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();

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

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

    nodeQueue.offer(root);
    levelQueue.offer(1); //start from 1

    while(!nodeQueue.isEmpty()){
        TreeNode node = nodeQueue.poll();
        int level = levelQueue.poll();

        List<Integer> l=null;
        if(result.size()<level){
            l = new ArrayList<>();
            result.add(l);
        }else{
            l = result.get(level-1);
        }

        l.add(node.val);

        if(node.left!=null){
            nodeQueue.offer(node.left);
            levelQueue.offer(level+1);
        }

        if(node.right!=null){
            nodeQueue.offer(node.right);
            levelQueue.offer(level+1);
        }
    }

    return result;
}

```

---

## 154 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]]

### 154.1 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;
}
```

---

# 155 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).

## 155.1 Java Solution

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

```
public List<List<Integer>> verticalOrder(TreeNode root) {  
    List<List<Integer>> result = new ArrayList<List<Integer>>();  
    if(root==null)  
        return result;  
  
    // level and list  
    HashMap<Integer, ArrayList<Integer>> map = new HashMap<Integer, ArrayList<Integer>>();  
  
    LinkedList<TreeNode> queue = new LinkedList<TreeNode>();  
    LinkedList<Integer> level = new LinkedList<Integer>();  
  
    queue.offer(root);  
    level.offer(0);  
  
    int minLevel=0;  
    int maxLevel=0;  
  
    while(!queue.isEmpty()) {  
        TreeNode p = queue.poll();  
        int l = level.poll();  
  
        //track min and max levels  
        minLevel=Math.min(minLevel, l);  
        maxLevel=Math.max(maxLevel, l);  
  
        if(map.containsKey(l)){  
            map.get(l).add(p.val);  
        }else{  
            ArrayList<Integer> list = new ArrayList<Integer>();  
            list.add(p.val);  
            map.put(l, list);  
        }  
  
        if(p.left!=null){  
            queue.offer(p.left);  
            level.offer(l-1);  
        }  
  
        if(p.right!=null){  
            queue.offer(p.right);  
            level.offer(l+1);  
        }  
    }  
    for(int i=minLevel; i<=maxLevel; i++)  
        result.add(map.get(i));  
    return result;  
}
```

```
        }
    }

    for(int i=minLevel; i<=maxLevel; i++){
        if(map.containsKey(i)){
            result.add(map.get(i));
        }
    }

    return result;
}
```

---

Time complexity is  $O(n)$  and space complexity is  $O(n)$ .  $n$  is the number of nodes on the tree.

# 156 Invert Binary Tree

## 156.1 Java Solution 1 - Recursive

---

```
public TreeNode invertTree(TreeNode root) {  
    helper(root);  
    return root;  
}  
  
public void helper(TreeNode n){  
    if(n==null){  
        return;  
    }  
  
    TreeNode t = n.left;  
    n.left = n.right;  
    n.right = t;  
  
    helper(n.left);  
    helper(n.right);  
}
```

---

## 156.2 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)  
            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;  
}
```

---

# 157 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)

## 157.1 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;
}
```

Similarly, we can also write the inorder traversal as the following:

```
public int kthSmallest(TreeNode root, int k) {
    Stack<TreeNode> stack = new Stack<TreeNode>();
    TreeNode p = root;
    while(p!=null){
        stack.push(p);
        p=p.left;
    }
    int i=0;
    while(!stack.isEmpty()){
        TreeNode t = stack.pop();
        i++;

        if(i==k)
            return t.val;

        TreeNode r = t.right;
        while(r!=null){
            stack.push(r);
            r=r.left;
        }
    }
}
```

```
    r=r.left;
}
return -1;
}
```

---

## 157.2 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  $O(\log(n))$ .

# 158 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).

## 158.1 Java Solution 1 - BFS

```
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);  
        }  
    }  
}
```

```
    }

    return max;
}
```

## 158.2 Java Solution 2 - DFS

```
class Solution {
    int max;

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

    private int helper(TreeNode t){
        if(t==null){
            return 0;
        }

        int leftMax = helper(t.left);
        int rightMax = helper(t.right);

        int leftTotal = 0;
        if(t.left == null){
            leftTotal = 1;
        }else if(t.val+1 == t.left.val){
            leftTotal = leftMax+1;
        }else{
            leftTotal = 1;
        }

        int rightTotal = 0;
        if(t.right == null){
            rightTotal = 1;
        }else if(t.val+1 == t.right.val){
            rightTotal = rightMax+1;
        }else{
            rightTotal = 1;
        }

        max = Math.max(max, leftTotal);
        max = Math.max(max, rightTotal);

        int longer = Math.max(leftTotal, rightTotal);

        return longer;
    }
}
```

# 159 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.

## 159.1 Java Solution 1 - Recursive

All values on the left sub tree must be less than parent and parent's parent, and all values on the right sub tree must be greater than parent and parent's parent. 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 time  $O(n)$  and space  $O(h)$ .

The following solution can handle violations close to root node faster.

```
public boolean isValidBST(TreeNode root) {  
    return helper(root, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY);  
}  
  
public boolean helper(TreeNode root, double min, double max){  
    if(root==null){  
        return true;  
    }  
  
    if(root.val<=min||root.val>=max){  
        return false;  
    }  
  
    boolean isLeftBST = helper(root.left, min, root.val);  
    boolean isRightBST = helper(root.right, root.val, max);  
  
    if(!isLeftBST||!isRightBST){  
        return false;  
    }
```

---

```

    }
    return true;
}

```

---

## 159.2 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;
    }
}

```

---

Time and space are both  $O(n)$ .

## 159.3 Java Solution 3 - In-order traversal

Since inorder traversal of BST is ascending, so we can check the sequence. Time is  $O(n)$  and space is  $O(h)$ .  $h$  is the height of the stack which is the tree's height.

# 160 Flatten Binary Tree to Linked List

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

For example, Given

---

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

---

The flattened tree should look like:

---

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

---

## 160.1 Java Solution

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

---

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

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

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

            if(p.left != null){
```

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

p = p.right;
}
}
```

---

# 161 Path Sum

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

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

---

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

---

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

## 161.1 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 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;
}
```

---

## 161.2 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);
}
```

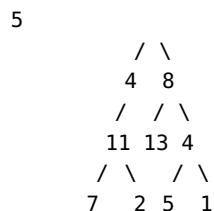
---

Thanks to nebulaliang, this solution is wonderful!

## 162 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]  
]
```

---

### 162.1 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.

### 162.2 Java Solution

---

```
public List<ArrayList<Integer>> pathSum(TreeNode root, int sum) {  
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();  
    if(root == null)  
        return result;  
  
    ArrayList<Integer> l = new ArrayList<Integer>();  
    l.add(root.val);  
    dfs(root, sum-root.val, result, l);  
    return result;  
}  
  
public void dfs(TreeNode t, int sum, ArrayList<ArrayList<Integer>> result, ArrayList<Integer> l){  
    if(t.left==null && t.right==null && sum==0){  
        ArrayList<Integer> temp = new ArrayList<Integer>();  
        temp.addAll(l);  
        result.add(temp);  
    }  
  
    //search path of left node  
    if(t.left != null){  
        l.add(t.left.val);
```

```
dfs(t.left, sum-t.left.val, result, l);
l.remove(l.size()-1);
}

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

---

# 163 Construct Binary Tree from Inorder and Postorder Traversal

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

## 163.1 Analysis

This problem can be illustrated by using a simple example.

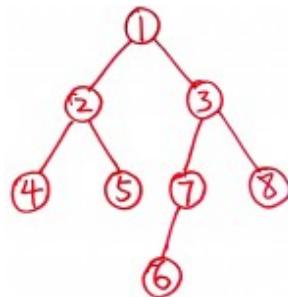
---

```
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.



## 163.2 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) {
    if (inStart > inEnd || postStart > postEnd)
        return null;

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

    int index = inStart;
    while (inorder[index] != rootValue)
        index++;

    root.left = buildTree(inorder, inStart, index - 1, postorder, postStart, postEnd - 1);
    root.right = buildTree(inorder, index + 1, inEnd, postorder, postEnd - 1, postEnd - 1);
    return root;
}
```

```
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 needs 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;
}
```

---

# 164 Construct Binary Tree from Preorder and Inorder Traversal

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

## 164.1 Analysis

Consider the following example:

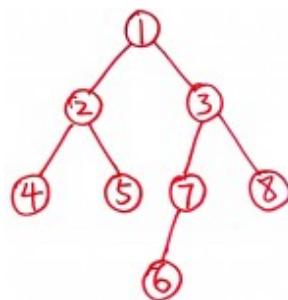
---

```
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.



## 164.2 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;
}
```

---

# 165 Convert Sorted Array to Binary Search Tree

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

## 165.1 Java Solution

A typical DFS problem using recursion.

---

```
// 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;
    }
}
```

---

# 166 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.

## 166.1 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.

## 166.2 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;
}
}
```

---

# 167 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.

## 167.1 Thoughts

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

## 167.2 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;
            }

            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;
}
```

---

# 168 Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree. For example, given the below binary tree

---

```
1
 / \
2  3
```

---

the result is 6.

## 168.1 Analysis

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

## 168.2 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;
}
```

---

# 169 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.

## 169.1 Analysis

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

## 169.2 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;
    }
}
```

```
    }
```

---

# 170 Symmetric Tree

## 170.1 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
```

---

## 170.2 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)
        return false;

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

    return true;
}
```

---

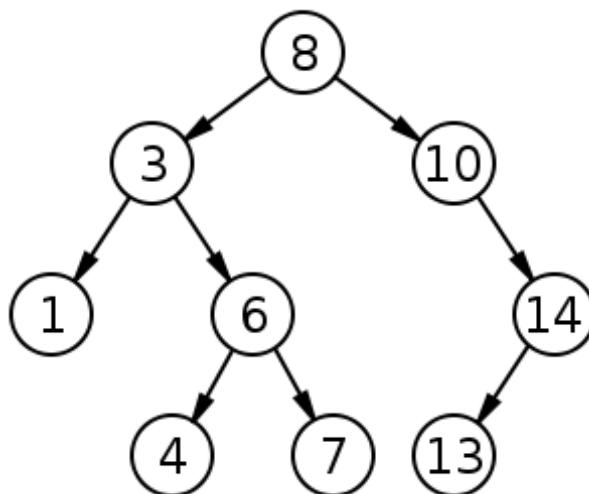
# 171 Binary Search Tree Iterator

## 171.1 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.

## 171.2 Java Solution

The key to solve this problem is understanding the features of BST. Here is an example BST.



---

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

    public int next() {
        if (stack.isEmpty())
            throw new NoSuchElementException("No elements");
        TreeNode top = stack.pop();
        int val = top.val;
        if (top.right != null)
            stack.push(top.right);
        return val;
    }

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

```
    }

    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;
    }
}
```

---

# 172 Binary Tree Right Side View

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

---

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

---

You can see [1, 3, 5].

## 172.1 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.

## 172.2 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){  
                queue.add(top.right);  
            }  
            //add left  
            if(top.left != null){  
                queue.add(top.left);  
            }  
        }  
    }  
}
```

---

```
    return result;  
}
```

---

# 173 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.

## 173.1 Analysis

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

## 173.2 Java Solution 1 - Recursive

---

```
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;  
}
```

---

## 173.3 Java Solution 2 - Iterative

---

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  
    TreeNode t = root;  
  
    while(t!=null){  
        if(p.val >t.val && q.val >t.val){  
            t = t.right;  
        }else if (p.val<t.val && q.val<t.val){  
            t = t.left;  
        }else{  
            return t;  
        }  
    }  
  
    return null;  
}
```

---

# 174 Lowest Common Ancestor of a Binary Tree

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

## 174.1 Java Solution 1

---

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if(root==null)
        return null;

    if(root==p || root==q)
        return root;

    TreeNode l = lowestCommonAncestor(root.left, p, q);
    TreeNode r = lowestCommonAncestor(root.right, p, q);

    if(l!=null&&r!=null){
        return root;
    }else if(l==null&&r==null){
        return null;
    }else{
        return l==null?r:l;
    }
}
```

---

To calculate time complexity, we know that  $f(n)=2*f(n-1)=2^2*f(n-2)=2^{\hat{o}}(log n)$ , so time=O(n).

## 174.2 Java Solution 2

---

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        CounterNode n = helper(root, p, q);
        return n.node;
    }

    public CounterNode helper(TreeNode root, TreeNode p, TreeNode q){
        if(root==null){
            return new CounterNode(null, 0);
        }

        CounterNode left = helper(root.left, p, q);
        if(left.count==2){
            return left;
        }

        CounterNode right = helper(root.right, p, q);
        if(right.count==2){
            return right;
        }

        if(left.node==null && right.node==null){
            return new CounterNode(null, 1);
        }else if(left.node==null && right.node!=null){
            return new CounterNode(right.node, 2);
        }else if(left.node!=null && right.node==null){
            return new CounterNode(left.node, 2);
        }else{
            return new CounterNode(left.node, 3);
        }
    }
}
```

---

```
}

int c=left.count+right.count+(root==p?1:0)+(root==q?1:0);

return new CounterNode(root, c);

}

class CounterNode{
    public int count;
    public TreeNode node;

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

---

# 175 Most Frequent Subtree Sum

Given the root of a tree, you are asked to find the most frequent subtree sum. The subtree sum of a node is defined as the sum of all the node values formed by the subtree rooted at that node (including the node itself). So what is the most frequent subtree sum value? If there is a tie, return all the values with the highest frequency in any order.

Examples 1 Input:

5 / 2 -3

return [2, -3, 4], since all the values happen only once, return all of them in any order.

Examples 2 Input:

5 / 2 -5

return [2], since 2 happens twice, however -5 only occur once.

## 175.1 Java Solution

We can solve this problem using a typical recursion on the tree, similar to [LCA](#).

```
public int[] findFrequentTreeSum(TreeNode root) {
    HashMap<Integer, Integer> map = new HashMap<>();

    helper(root, map);

    int maxCount = 0;
    for(int i: map.keySet()){
        if(map.get(i)>maxCount){
            maxCount=map.get(i);
        }
    }

    List<Integer> mf = new ArrayList<>();
    for(Map.Entry<Integer, Integer> entry: map.entrySet()){
        if(entry.getValue()==maxCount){
            mf.add(entry.getKey());
        }
    }

    int[] result = new int[mf.size()];
    int k=0;
    for(int i: mf){
        result[k++]=i;
    }

    return result;
}

public Integer helper(TreeNode root, HashMap<Integer, Integer> map){
    if(root==null){
        return null;
    }
}
```

```
Integer left = helper(root.left, map);
if(left==null){
    left=0;
}

Integer right = helper(root.right, map);
if(right==null){
    right=0;
}

int sum = root.val + left + right;
map.put(sum, map.getOrDefault(sum, 0)+1);

return sum;
}
```

---

## 176 Verify Preorder Serialization of a Binary Tree

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

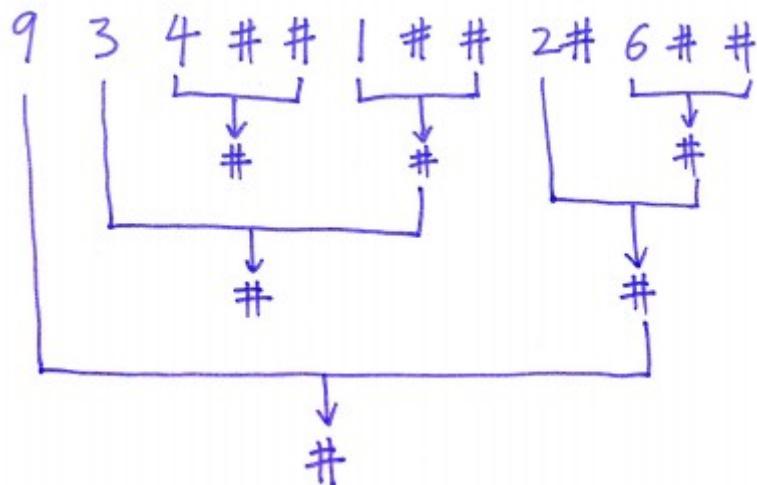
```
9
 / \
3   2
/ \ / \
4 1 # 6
/ \ / \ / \
# # # # # #
```

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

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

### 176.1 Java Solution

We can keep trimming the leaves until there is no one to remove. If a sequence is like "4 # #", change it to "#" and continue. A stack is a good data structure for this purpose.



```
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
            if(stack.get(0).equals("#") && stack.get(1).equals("#") && stack.get(2).equals("#")){
                stack.remove(0);
                stack.remove(1);
                stack.remove(2);
            }
        }
    }
    return stack.size() == 1 && stack.get(0).equals("#");
}
```

```

    && 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;
}

```

---

If only stack operations are allowed, the solution can be written in the following way:

```

public boolean isValidSerialization(String preorder) {
    String[] arr = preorder.split(",");
    Stack<String> stack = new Stack<>();
    for(String s: arr){
        if(stack.isEmpty() || !s.equals("#")){
            stack.push(s);
        }else{
            while(!stack.isEmpty() && stack.peek().equals("#")){
                stack.pop();
                if(stack.isEmpty()){
                    return false;
                }else{
                    stack.pop();
                }
            }
            stack.push("#");
        }
    }

    return stack.size()==1 && stack.peek().equals("#");
}

```

---

# 177 Populating Next Right Pointers in Each Node

Given the following perfect binary tree,

---

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

---

After calling your function, the tree should look like:

---

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

---

## 177.1 Java Solution 1 - Simple

---

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

    LinkedList<TreeLinkNode> nodeQueue = new LinkedList<TreeLinkNode>();
    LinkedList<Integer> depthQueue = new LinkedList<Integer>();

    if(root!=null){
        nodeQueue.offer(root);
        depthQueue.offer(1);
    }

    while(!nodeQueue.isEmpty()){
        TreeLinkNode topNode = nodeQueue.poll();
        int depth = depthQueue.poll();

        if(depthQueue.isEmpty()){
            topNode.next = null;
        }else if(depthQueue.peek()>depth){
            topNode.next = null;
        }else{
            topNode.next = nodeQueue.peek();
        }

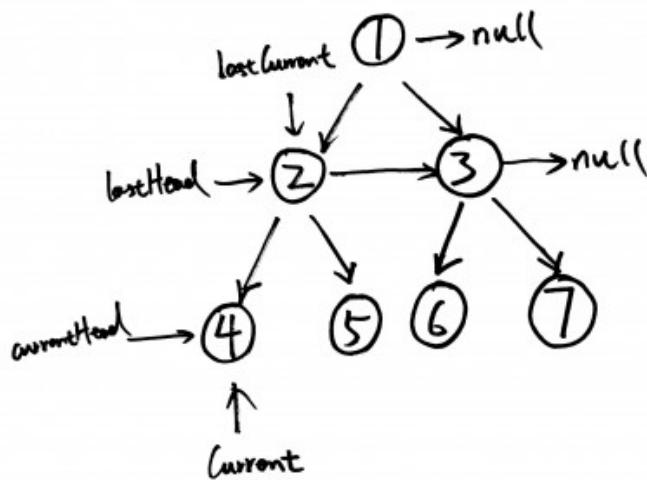
        if(topNode.left!=null){
            nodeQueue.offer(topNode.left);
            depthQueue.offer(depth+1);
        }

        if(topNode.right!=null){
```

```
        nodeQueue.offer(topNode.right);
        depthQueue.offer(depth+1);
    }
}
```

## 177.2 Java Solution 2

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) {  
    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;
}

}
```

---

# 178 Populating Next Right Pointers in Each Node

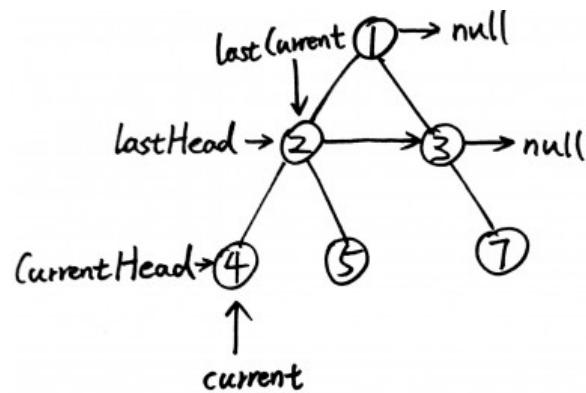
II

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?

## 178.1 Analysis

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



## 178.2 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) {  
                if(currentHead == null){  
                    currentHead = lastCurrent.left;  
                    current = lastCurrent.left;  
                }else{  
                    current.next = lastCurrent.left;  
                    current = current.next;  
                }  
            }  
            lastCurrent = lastCurrent.right;  
        }  
        lastHead = lastCurrent;  
    }  
}
```

```
        }

    //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;
}

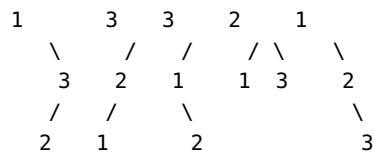
//update last head
lastHead = currentHead;
currentHead = null;
}
}
```

---

# 179 Unique Binary Search Trees

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

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



## 179.1 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.

## 179.2 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];
}
```

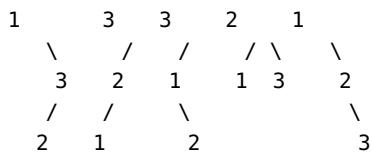
---

Check out [how to get all unique binary search trees](#).

# 180 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.



## 180.1 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.

## 180.2 Java Solution

---

```
public List<TreeNode> generateTrees(int n) {
    if(n==0){
        return new ArrayList<TreeNode>();
    }

    return helper(1, n);
}

public List<TreeNode> helper(int m, int n){
    List<TreeNode> result = new ArrayList<TreeNode>();
    if(m>n){
        result.add(null);
        return result;
    }

    for(int i=m; i<=n; i++){
        List<TreeNode> ls = helper(m, i-1);
        List<TreeNode> rs = helper(i+1, n);
        for(TreeNode l: ls){
            for(TreeNode r: rs){
                TreeNode curr = new TreeNode(i);
                curr.left=l;
                curr.right=r;
                result.add(curr);
            }
        }
    }
}

return result;
```

}

# 181 Sum Root to Leaf Numbers

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

For example,

---

```
1
 / \
2  3
```

---

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

## 181.1 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;  
}
```

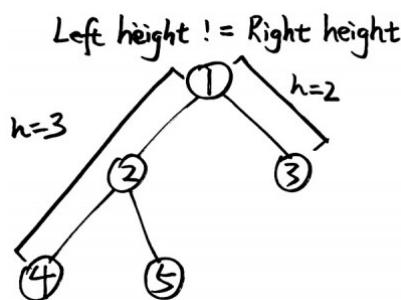
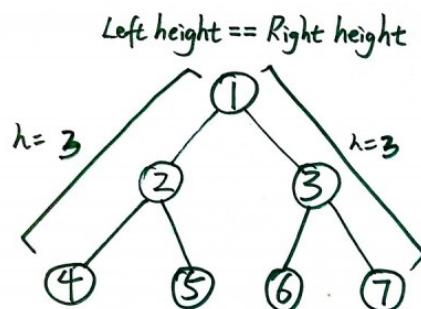
---

## 182 Count Complete Tree Nodes

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

### 182.1 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  $O(h^2)$ .

### 182.2 Java Solution

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

    int left = getLeftHeight(root)+1;
    int right = getRightHeight(root)+1;

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

```
    }

}

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

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

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

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

---

# 183 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.

## 183.1 Java Solution 1 - Recursion

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);
        }
    }
}
```

---

## 183.2 Java Solution 2 - Iteration

---

```
public int closestValue(TreeNode root, double target) {
    double min=Double.MAX_VALUE;
    int result = root.val;

    while(root!=null){
        if(target>root.val){

            double diff = Math.abs(root.val-target);
            if(diff<min){
                min = Math.min(min, diff);
                result = root.val;
            }
        }
    }
}
```

---

```
        }
        root = root.right;
    }else if(target<root.val){

        double diff = Math.abs(root.val-target);
        if(diff<min){
            min = Math.min(min, diff);
            result = root.val;
        }
        root = root.left;
    }else{
        return root.val;
    }
}

return result;
}
```

---

# 184 Binary Tree Paths

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

## 184.1 Naive DFS 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);
    }

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

---

## 184.2 Simplified DFS Solution

This depth-first solution can be much simplified like the following:

---

```
public List<String> binaryTreePaths(TreeNode root) {  
  
    String sb = "";  
    ArrayList<String> result = new ArrayList<String>();  
  
    helper(root, result, sb);  
  
    return result;  
}  
  
public void helper(TreeNode root, ArrayList<String> result, String s){  
    if(root==null){  
        return;  
    }  
  
    s = s+"->"+root.val;  
  
    if(root.left==null &&root.right==null){  
        result.add(s.substring(2));  
        return;  
    }  
  
    if(root.left!=null){  
        helper(root.left, result, s);  
    }  
    if(root.right!=null){  
        helper(root.right, result, s);  
    }  
}
```

---

# 185 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.

## 185.1 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;  
}
```

---

# 186 Recover Binary Search Tree

Two elements of a binary search tree (BST) are swapped by mistake. Recover the tree without changing its structure.

## 186.1 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){
            int val = second.val;
            second.val = first.val;
            first.val = val;
        }
    }
}
```

---

## 187 Same Tree

Two binary trees are considered the same if they have identical structure and nodes have the same value.  
This problem can be solved by using a simple recursive function.

---

```
public boolean isSameTree(TreeNode p, TreeNode q) {  
    if(p==null && q==null){  
        return true;  
    }else if(p==null || q==null){  
        return false;  
    }  
  
    if(p.val==q.val){  
        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);  
    }else{  
        return false;  
    }  
}
```

---

# 188 Serialize and Deserialize Binary Tree

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

## 188.1 Java Solution 1 - Level Order Traversal

---

```
// Encodes a tree to a single string.
public String serialize(TreeNode root) {
    if(root==null){
        return "";
    }

    StringBuilder sb = new StringBuilder();

    LinkedList<TreeNode> queue = new LinkedList<TreeNode>();

    queue.add(root);
    while(!queue.isEmpty()){
        TreeNode t = queue.poll();
        if(t!=null){
            sb.append(String.valueOf(t.val) + ",");
            queue.add(t.left);
            queue.add(t.right);
        }else{
            sb.append("#,");
        }
    }

    sb.deleteCharAt(sb.length()-1);
    System.out.println(sb.toString());
    return sb.toString();
}

// Decodes your encoded data to tree.
public TreeNode deserialize(String data) {
    if(data==null || data.length()==0)
        return null;

    String[] arr = data.split(",");
    TreeNode root = new TreeNode(Integer.parseInt(arr[0]));

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

    int i=1;
    while(!queue.isEmpty()){
        TreeNode t = queue.poll();
    }
```

```

if(t==null)
    continue;

if(!arr[i].equals("#")){
    t.left = new TreeNode(Integer.parseInt(arr[i]));
    queue.offer(t.left);
}

}else{
    t.left = null;
    queue.offer(null);
}

i++;

if(!arr[i].equals("#")){
    t.right = new TreeNode(Integer.parseInt(arr[i]));
    queue.offer(t.right);
}

else{
    t.right = null;
    queue.offer(null);
}
i++;
}

return root;
}

```

---

## 188.2 Java Solution 2 - Preorder Traversal

```

// Encodes a tree to a single string.
public String serialize(TreeNode root) {
    if(root==null)
        return null;

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

    while(!stack.isEmpty()){
        TreeNode h = stack.pop();
        if(h!=null){
            sb.append(h.val+",");
            stack.push(h.right);
            stack.push(h.left);
        }else{
            sb.append("#,");
        }
    }

    return sb.toString().substring(0, sb.length()-1);
}

// Decodes your encoded data to tree.
public TreeNode deserialize(String data) {

```

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

int[] t = {0};
String[] arr = data.split(",");
return helper(arr, t);
}

public TreeNode helper(String[] arr, int[] t){
    if(arr[t[0]].equals("#")){
        return null;
    }

    TreeNode root = new TreeNode(Integer.parseInt(arr[t[0]]));

    t[0]=t[0]+1;
    root.left = helper(arr, t);
    t[0]=t[0]+1;
    root.right = helper(arr, t);

    return root;
}
```

---

# 189 Inorder Successor in BST

Given a binary search tree and a node in it, find the in-order successor of that node in the BST.

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

## 189.1 Java Solution

The node does not have a pointer pointing to its parent. This is different from [Inorder Successor in BST II](#).

```
public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
    if(root==null)
        return null;

    TreeNode next = null;
    TreeNode c = root;
    while(c!=null && c.val!=p.val){
        if(c.val > p.val){
            next = c;
            c = c.left;
        }else{
            c= c.right;
        }
    }

    if(c==null)
        return null;

    if(c.right==null)
        return next;

    c = c.right;
    while(c.left!=null)
        c = c.left;

    return c;
}
```

When the tree is balanced, time complexity is  $O(\log(n))$  and space is  $O(1)$ . The worst case time complexity is  $O(n)$ .

## 190 Inorder Successor in BST II

Given a binary search tree and a node in it, find the in-order successor of that node in the BST.

The successor of a node p is the node with the smallest key greater than p.val. You will have direct access to the node but not to the root of the tree. Each node will have a reference to its parent node. A node is defined as the following:

---

```
// Definition for a Node.
class Node {
    public int val;
    public Node left;
    public Node right;
    public Node parent;
};

*/
```

---

### 190.1 Java Solution

---

```
public Node inorderSuccessor(Node x) {
    Node result = null;

    //case 1: right child is not null -> go down to get the next
    Node p = x.right;
    while(p!=null){
        result = p;
        p = p.left;
    }

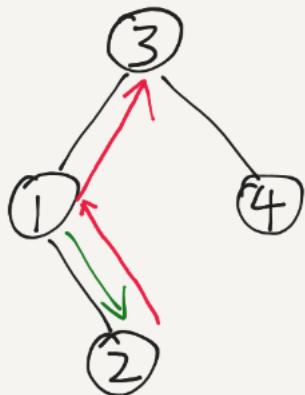
    if(result != null){
        return result;
    }

    //case 2: right child is null -> go up to the parent,
    //until the node is a left child, return the parent
    p = x;

    while(p!=null){
        if(p.parent!=null && p.parent.left==p){
            return p.parent;
        }
        p = p.parent;
    }

    return null;
}
```

---



case 1: target is ①  
go down to get next

case 2: target is ②  
go up to get the next  
next is the parent reached  
from the left child

If the tree is balanced, the time complexity is the height of the tree -  $O(\log(n))$ . In the worst cast, the time is  $O(n)$ . Space complexity is constant.

# 191 Find Leaves of Binary Tree

Given a binary tree, collect a tree's nodes as if you were doing this: Collect and remove all leaves, repeat until the tree is empty.

Example: Given binary tree

---

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

---

Returns [4, 5, 3], [2], [1].

## 191.1 Java Solution 1

Naively, we can get the order of each node, store them in a hashmap and then iterate over the hashmap to get the list.

```
public List<List<Integer>> findLeaves(TreeNode root) {
    HashMap<TreeNode, Integer> map=new HashMap<>();
    helper(root, map);

    int min = Integer.MAX_VALUE;
    int max = Integer.MIN_VALUE;
    HashMap<Integer, HashSet<TreeNode>> reversed = new HashMap<>();

    for(Map.Entry<TreeNode, Integer> entry: map.entrySet()){
        min = Math.min(min, entry.getValue());
        max = Math.max(max, entry.getValue());

        HashSet<TreeNode> set = reversed.getOrDefault(entry.getValue(), new HashSet<TreeNode>());
        set.add(entry.getKey());
        reversed.put(entry.getValue(), set);
    }

    List<List<Integer>> result = new ArrayList<List<Integer>>();
    for(int i=min; i<=max; i++){
        HashSet<TreeNode> set = reversed.get(i);
        ArrayList<Integer> l = new ArrayList<>();
        for(TreeNode td: set){
            l.add(td.val);
        }
        result.add(l);
    }

    return result;
}

private int helper(TreeNode root, HashMap<TreeNode, Integer> map){
```

---

```

if(root==null){
    return 0;
}

int left = helper(root.left, map);
int right = helper(root.right, map);

int order = Math.max(left, right)+1;
map.put(root, order);
return order;
}

```

---

## 191.2 Java Solution 2 - Optimized

The key to solve this problem is converting the problem to be finding the index of the element in the result list. Then this is a typical DFS problem on trees.

---

```

public List<List<Integer>> findLeaves(TreeNode root) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    helper(result, root);
    return result;
}

// traverse the tree bottom-up recursively
private int helper(List<List<Integer>> list, TreeNode root){
    if(root==null)
        return -1;

    int left = helper(list, root.left);
    int right = helper(list, root.right);
    int curr = Math.max(left, right)+1;

    // the first time this code is reached is when curr==0,
    // since the tree is bottom-up processed.
    if(list.size()<=curr){
        list.add(new ArrayList<Integer>());
    }

    list.get(curr).add(root.val);

    return curr;
}

```

---

# 192 Largest BST Subtree

Given a binary tree, find the largest subtree which is a Binary Search Tree (BST), where largest means subtree with largest number of nodes in it.

## 192.1 Java Solution

```
class Wrapper{
    int size;
    int lower, upper;
    boolean isBST;

    public Wrapper(){
        lower = Integer.MAX_VALUE;
        upper = Integer.MIN_VALUE;
        isBST = false;
        size = 0;
    }
}

public class Solution {
    public int largestBSTSubtree(TreeNode root) {
        return helper(root).size;
    }

    public Wrapper helper(TreeNode node){
        Wrapper curr = new Wrapper();

        if(node == null){
            curr.isBST= true;
            return curr;
        }

        Wrapper l = helper(node.left);
        Wrapper r = helper(node.right);

        //current subtree's boundaries
        curr.lower = Math.min(node.val, l.lower);
        curr.upper = Math.max(node.val, r.upper);

        //check left and right subtrees are BST or not
        //check left's upper again current's value and right's lower against current's value
        if(l.isBST && r.isBST && l.upper<=node.val && r.lower>=node.val){
            curr.size = l.size+r.size+1;
            curr.isBST = true;
        }else{
            curr.size = Math.max(l.size, r.size);
            curr.isBST = false;
        }

        return curr;
    }
}
```

```
    }  
}
```

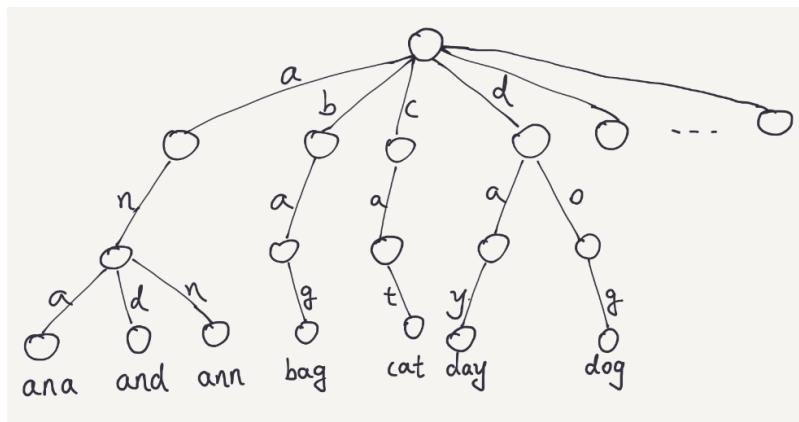
---

# 193 Implement Trie (Prefix Tree)

Implement a [trie](#) with insert, search, and startsWith methods.

## 193.1 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 the trie in the following diagram to walk though the Java solution.



---

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

    public TrieNode() {}

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

public class Trie {
    private TrieNode root;

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

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

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

---

```

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

children = t.children;

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

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

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

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

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

    return t;
}
}

```

## 193.2 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;
    }

    // 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;
}
```

---

If the same words can be inserted more than once, what do you need to change to make it work?

# 194 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.

## 194.1 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;
    }
}
}

```

## 194.2 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;
    }
}

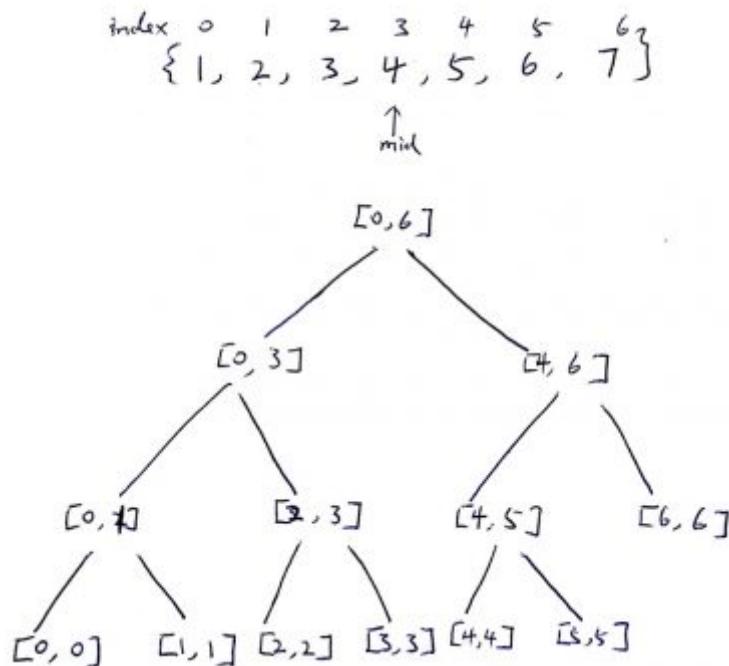
return false;
}
```

---

# 195 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  $val$ .

## 195.1 Java Solution 1 - Segment Tree



---

```
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;
        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;
}
}

```

---

## 195.2 Java Solution 2 - Binary Index Tree / Fenwick Tree

Here is a perfect video to show how binary index tree works. In addition, my notes at the end of the post may also help.

---

```

public class NumArray {

    int[] btree;
    int[] arr;

    public NumArray(int[] nums) {
        btree = new int[nums.length+1];
        arr = nums;

        for(int i=0; i<nums.length; i++){
            add(i+1, nums[i]);
        }
    }

    void add(int i, int val) {
        for(int j=i; j<btree.length; j=j+(j&(-j)) ){
            btree[j] += val;
        }
    }

    void update(int i, int val) {
        add(i+1, val-arr[i]);
        arr[i]=val;
    }

    public int sumRange(int i, int j) {
        return sumHelper(j+1)-sumHelper(i);
    }

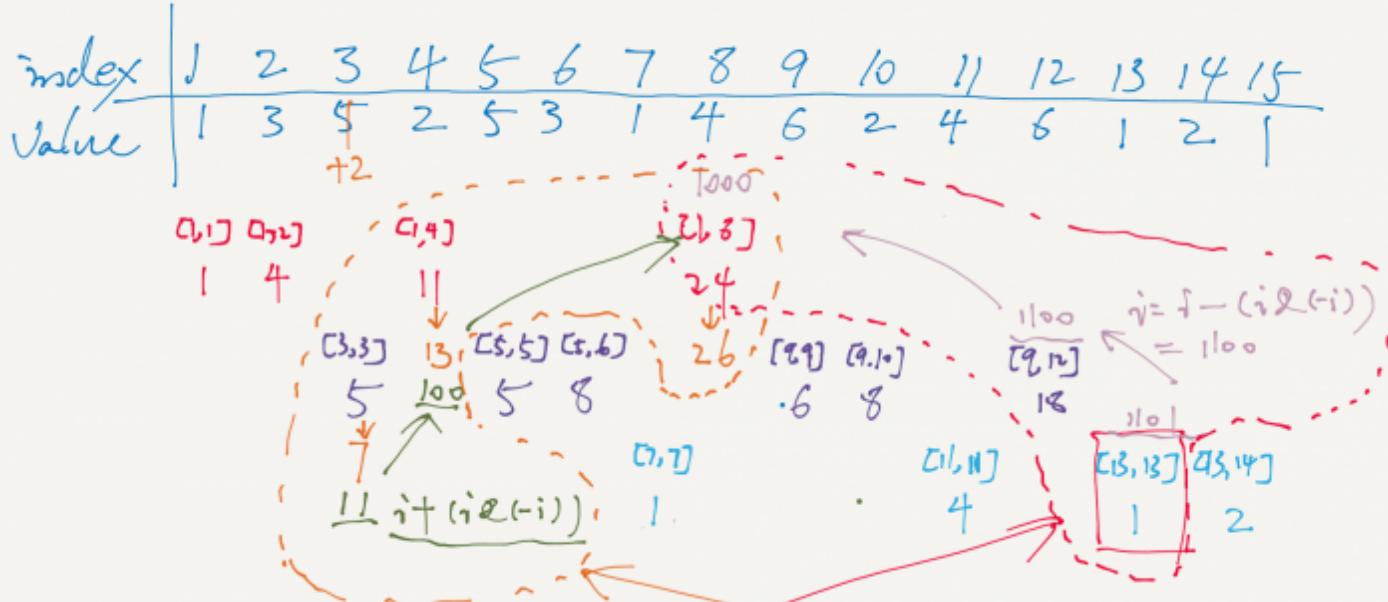
    public int sumHelper(int i){
        int sum=0;
        for(int j=i; j>=1; j=j-(j&(-j))){
            sum += btree[j];
        }
    }
}

```

```

    }
    return sum;
}

```



# 196 The Skyline Problem

## 196.1 Analysis

This problem is essentially a problem of processing  $2n$  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.

## 196.2 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;
}
```

# 197 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.

## 197.1 Analysis

This problem can be solved by using two queues.

## 197.2 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--;
                }
            }
        }
    }

    // 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();
}
```

---

# 198 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.

## 198.1 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();  
    }  
}
```

---

# 199 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 accepts 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.

## 199.1 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 true;
    }

    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
```

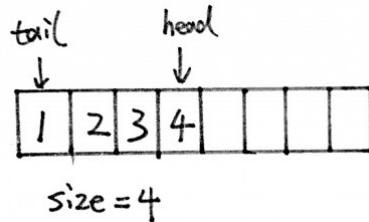
---

This example is used 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.

You may check out [how to implement a queue by using an array](#).

## 200 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;
        }

        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);
}
}
```

---

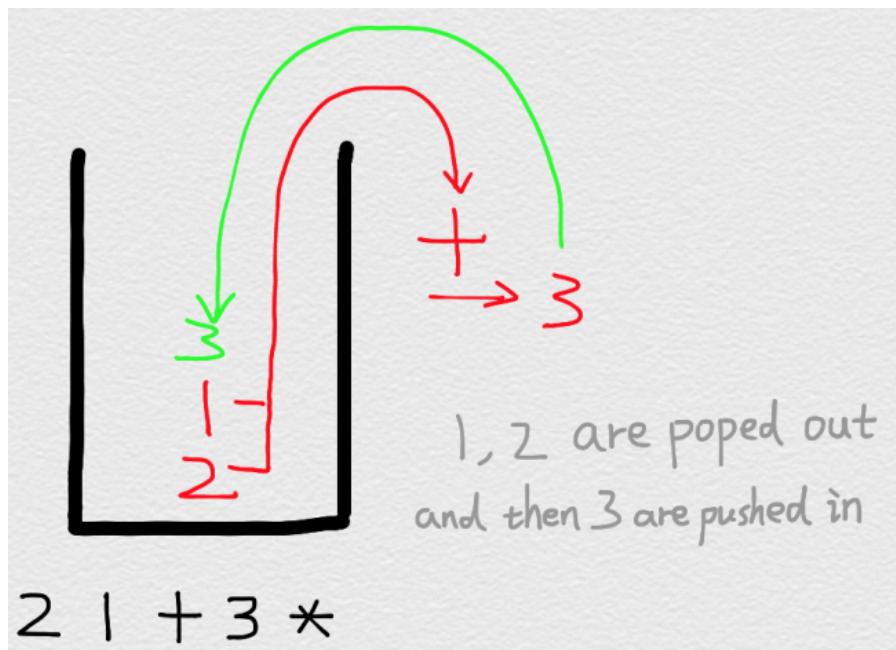
# 201 Evaluate Reverse Polish Notation

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
```

## 201.1 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));  
    }  
  
    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.

## 201.2 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;
}
}
```

---

# 202 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.

## 202.1 Analysis

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

## 202.2 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();
}
```

---

# 203 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.

## 203.1 Java Solution

A stack can be used to track and reduce pairs. Since the problem asks for length, we can put the index into the stack along with the character. For coding simplicity purpose, we can use 0 to represent ( and 1 to represent ).

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

---

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

    for(int i=0; i<s.length(); i++){
        char c = s.charAt(i);
        if(c=='')){
            if(!stack.isEmpty() && stack.peek()[0]==0){
                stack.pop();
                result = Math.max(result, i-(stack.isEmpty()?-1:stack.peek()[1]));
            }else{
                stack.push(new int[]{1, i});
            }
        }else{
            stack.push(new int[]{0, i});
        }
    }

    return result;
}
```

---

# 204 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.

## 204.1 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.

## 204.2 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(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;
    }
}
}

```

---

## 204.3 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.pop() != s.charAt(index))
            return false;
        index++;
    }

    return true;
}

```

---

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

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

return true;
}
```

---

## 204.4 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);

        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));
    }
}
```

---

# 205 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.

## 205.1 Java Solution

To make constant time of getMin(), we need to keep track of the minimum element for each element in the stack. Define an element class that holds element value, min value, and pointer to elements below it.

```
class Elem{
    public int value;
    public int min;
    public Elem next;

    public Elem(int value, int min){
        this.value = value;
        this.min = min;
    }
}

public class MinStack {
    public Elem top;

    /** initialize your data structure here. */
    public MinStack() {

    }

    public void push(int x) {
        if(top == null){
            top = new Elem(x, x);
        }else{
            Elem e = new Elem(x, Math.min(x,top.min));
            e.next = top;
            top = e;
        }
    }

    public void pop() {
        if(top == null)
            return;
        Elem temp = top.next;
        top.next = null;
        top = temp;
    }

    public int top() {
```

```
    if(top == null)
        return -1;
    return top.value;
}

public int getMin() {
    if(top == null)
        return -1;
    return top.min;
}
}
```

---

# 206 Max Chunks To Make Sorted

Given an array arr that is a permutation of  $[0, 1, \dots, \text{arr.length} - 1]$ , we split the array into some number of "chunks" (partitions), and individually sort each chunk. After concatenating them, the result equals the sorted array.

What is the most number of chunks we could have made?

For example, given  $[2,0,1]$ , the method returns 0, as there can only be one chunk.

## 206.1 Analysis

The key to solve this problem is using a stack to track the existing chunk. Each chunk is represented a min and max number. Each chunk is essentially an interval and the interval can not overlap.

## 206.2 Java Solution

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

    // use [min,max] for each chunk
    Stack<int[]> stack = new Stack<int[]>();

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

        while(!stack.isEmpty()){
            int[] top = stack.peek();

            if(arr[i] < top[1]){
                min=Math.min(top[0], min);
                max=Math.max(max, top[1]);
                stack.pop();
            }else{
                break;
            }
        }

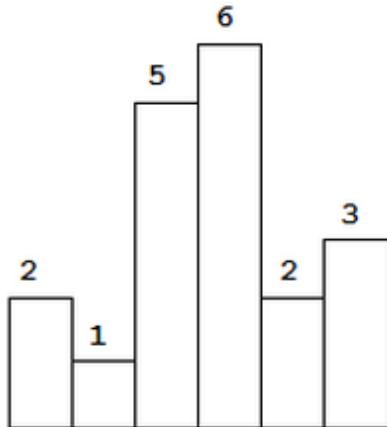
        stack.push(new int[]{min,max});
    }

    return stack.size();
}
```

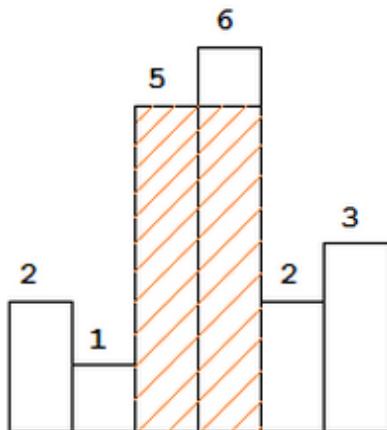
Time complexity is  $O(n)$ .

## 207 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.

### 207.1 Analysis

If a bar is blocked by a lower bar, then the taller bar is no need to be considered any more. We only need to keep track of the bars that are not blocked. As we iterate over the bars, whenever a bar blocks a previous bar, we calculate how much area the previous bar can support.

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

## 207.2 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;
}
```

---

# 208 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.

## 208.1 Analysis

This problem can be converted to the "Largest Rectangle in Histogram" problem.

## 208.2 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;

    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;  
}
```

---

## 209 Mini Parser

**209.1 Given a nested list of integers represented as a string, implement a parser to deserialize it. Each element is either an integer, or a list – whose elements may also be integers or other lists.**

Note: You may assume that the string is well-formed:

- String is non-empty.
- String does not contain white spaces.
- String contains only digits and "[],"

For example,

---

Given s = "[123,[456,[789]]]",

Return a NestedInteger object containing a nested list with 2 elements:

1. An integer containing value 123.
  2. A nested list containing two elements:
    - i. An integer containing value 456.
    - ii. A nested list with one element:
      - a. An integer containing value 789.
- 

## 209.2 Java Solution

To solve this problem, we should add more example to make clear what is the expected output. For example, s = "[123,[456],789]" is a legal input.

---

```
public NestedInteger deserialize(String s) {
    Stack<NestedInteger> stack = new Stack<>();
    StringBuilder sb = new StringBuilder();

    for(int i=0; i<s.length(); i++){
        char c = s.charAt(i);
        switch(c){
            case '[':
                NestedInteger ni = new NestedInteger();
                stack.push(ni);
                break;

            case ']':
                if(sb.length()>0){ //123, not [456],
                    stack.peek().add(new NestedInteger(Integer.parseInt(sb.toString())));
                    sb.delete(0, sb.length());
                }
        }
    }
}
```

```
NestedInteger top = stack.pop();
if(stack.isEmpty()){
    return top;
}else{
    stack.peek().add(top);
}

break;
case ',':
if(sb.length()>0){ //handle case "123," not "[456],"
    stack.peek().add(new NestedInteger(Integer.parseInt(sb.toString())));
    sb=sb.delete(0, sb.length());
}

break;

default: //digits
sb.append(c);
}
}

//handle case "123"
if(sb.length()>0){
    return new NestedInteger(Integer.parseInt(sb.toString()));
}

return null;
}
```

# 210 Flatten Nested List Iterator

Given a nested list of integers, implement an iterator to flatten it. Each element is either an integer, or a list – whose elements may also be integers or other lists.

For example, given the list `[[1,1],2,[1,1]]`, by calling next repeatedly until hasNext returns false, the order of elements returned by next should be: `[1,1,2,1,1]`.

## 210.1 Java Solution 1

---

```
public class NestedIterator implements Iterator<Integer> {
    Stack<NestedInteger> stack = new Stack<NestedInteger>();

    public NestedIterator(List<NestedInteger> nestedList) {
        if(nestedList==null)
            return;

        for(int i=nestedList.size()-1; i>=0; i--){
            stack.push(nestedList.get(i));
        }
    }

    @Override
    public Integer next() {
        return stack.pop().getInteger();
    }

    @Override
    public boolean hasNext() {
        while(!stack.isEmpty()){
            NestedInteger top = stack.peek();
            if(top.isInteger()){
                return true;
            }else{
                stack.pop();
                for(int i=top.getList().size()-1; i>=0; i--){
                    stack.push(top.getList().get(i));
                }
            }
        }
        return false;
    }
}
```

---

## 210.2 Java Solution 2

---

```
public class NestedIterator implements Iterator<Integer> {
```

```
Stack<Iterator<NestedInteger>> stack = new Stack<Iterator<NestedInteger>>();
Integer current;

public NestedIterator(List<NestedInteger> nestedList) {
    if(nestedList==null)
        return;

    stack.push(nestedList.iterator());
}

@Override
public Integer next() {
    Integer result = current;
    current = null;
    return result;
}

@Override
public boolean hasNext() {
    while(!stack.isEmpty() && current==null){
        Iterator<NestedInteger> top = stack.peek();
        if(!top.hasNext()){
            stack.pop();
            continue;
        }

        NestedInteger n = top.next();
        if(n.isInteger()){
            current = n.getInteger();
            return true;
        }else{
            stack.push(n.getList().iterator());
        }
    }

    return false;
}
}
```

# 211 Nested List Weight Sum

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list – whose elements may also be integers or other lists.

Example 1: Given the list [[1,1],2,[1,1]], return 10. (four 1's at depth 2, one 2 at depth 1)

## 211.1 Java Solution 1 - Recursive

---

```
public int depthSum(List<NestedInteger> nestedList) {
    return helper(nestedList, 1);
}

public int helper(List<NestedInteger> nestedList, int depth){
    if(nestedList==null||nestedList.size()==0)
        return 0;

    int sum=0;
    for(NestedInteger ni: nestedList){
        if(ni.isInteger()){
            sum += ni.getInteger() * depth;
        }else{
            sum += helper(ni.getList(), depth+1);
        }
    }

    return sum;
}
```

---

## 211.2 Java Solution 2 - Iterative

---

```
public int depthSum(List<NestedInteger> nestedList) {
    int sum=0;

    LinkedList<NestedInteger> queue = new LinkedList<NestedInteger>();
    LinkedList<Integer> depth = new LinkedList<Integer>();

    for(NestedInteger ni: nestedList){
        queue.offer(ni);
        depth.offer(1);
    }

    while(!queue.isEmpty()){
        NestedInteger top = queue.poll();
        int dep = depth.poll();

        if(top.isInteger()){
            sum += dep*top.getInteger();
        }
    }
}
```

---

```
    }else{
        for(NestedInteger ni: top.getList()){
            queue.offer(ni);
            depth.offer(dep+1);
        }
    }

    return sum;
}
```

---

## 212 Longest Absolute File Path

Suppose we abstract our file system by a string in the following manner:

The string "dir12 .ext" represents:

---

```
dir
  subdir1
  subdir2
    file.ext
```

---

The directory dir contains an empty sub-directory subdir1 and a sub-directory subdir2 containing a file file.ext.  
The string "dir1 1.ext 12 2 2.ext" represents:

---

```
dir
  subdir1
    file1.ext
    subsubdir1
  subdir2
    subsubdir2
      file2.ext
```

---

The directory dir contains two sub-directories subdir1 and subdir2. subdir1 contains a file file1.ext and an empty second-level sub-directory subsubdir1. subdir2 contains a second-level sub-directory subsubdir2 containing a file file2.ext.

We are interested in finding the longest (number of characters) absolute path to a file within our file system. For example, in the second example above, the longest absolute path is "dir/subdir2/subsubdir2/file2.ext", and its length is 32 (not including the double quotes).

Given a string representing the file system in the above format, return the length of the longest absolute path to file in the abstracted file system. If there is no file in the system, return 0.

Note:

- The name of a file contains at least a . and an extension.
- The name of a directory or sub-directory will not contain a ..
- Time complexity required: O(n) where n is the size of the input string.

Notice that a/aa/aaa/file1.txt is not the longest file path, if there is another pathaaaaaaaaaaaaaaaaaaa/sth.png.

### 212.1 Java Solution

---

```
class Node{
    int level;
    int len;
    public Node(int lev, int len){
        this.level = lev;
        this.len = len;
    }
}

public class Solution {
```

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

    int max=0;

    String[] arr = input.split("\n");

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

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

        int count=0;
        int j=0;
        while(j<s.length()-1){
            //System.out.println("first " + s.substring(j, j+2));
            if(s.substring(j, j+1).equals("\t")){
                j++;
                count++;
            }else{
                break;
            }
        }

        while(!stack.isEmpty() && count <=stack.peek().level){
            stack.pop();
        }

        if(s.contains(".")){
            if(stack.isEmpty()){
                max = Math.max(max, s.length()-count);
            }else{
                max = Math.max(max, stack.peek().len+s.length()-count);
            }
        }else{
            if(stack.isEmpty()){
                stack.push(new Node(count, s.length()-count+1));
            }else{
                stack.push(new Node(count, stack.peek().len + s.length()-count+1));
            }
        }
    }

    return max;
}
```

## 213 Decode String

Given an encoded string, return it's decoded string.

The encoding rule is:  $k[\text{encoded\_string}]$ , where the `encoded_string` inside the square brackets is being repeated exactly  $k$  times. Note that  $k$  is guaranteed to be a positive integer.

You may assume that the input string is always valid; No extra white spaces, square brackets are well-formed, etc.

Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers,  $k$ . For example, there won't be input like `3a` or `2[4]`.

Examples:

---

```
s = "3[a]2[bc]", return "aaabcbc".
s = "3[a2[c]]", return "accaccacc".
s = "2[abc]3[cd]ef", return "abcabccdcdef".
```

---

### 213.1 Java Solution

The key to solve this problem is convert the string to a structured data structure and recursively form the return string.

---

```
class Node{
    int num;
    ArrayList<Node> list;
    char symbol;
    boolean isList;

    public Node(char s){
        symbol=s;
    }

    public Node(int n){
        list = new ArrayList<Node>();
        isList=true;
        num=n;
    }

    public String toString(){
        String s = "";
        if(isList){
            s += num + ":" + list.toString();
        }else{
            s += symbol;
        }
        return s;
    }
}

public class Solution {
    public String decodeString(String s) {
        int i = 0;
```

---

```

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

stack.push(new Node(1));

String t = "";
while (i < s.length()) {
    char c = s.charAt(i);

    // new Node
    if (c >= '0' && c <= '9') {
        t += c;

    } else if (c == '[') {
        if (t.length() > 0) {
            int num = Integer.parseInt(t);
            stack.push(new Node(num));
            t = "";
        }
    } else if (c == ']') {
        Node top = stack.pop();

        if (stack.isEmpty()) {

        } else {
            stack.peek().list.add(top);
        }

        } else {
            stack.peek().list.add(new Node(c));
        }
    }

    i++;
}

return getString(stack.peek());
}

public String getString(Node node){
    String s="";
    if(node.isList){
        for(int i=0; i<node.num; i++){
            for(Node t: node.list)
                s+= getString(t);
        }
    }else{
        s+=node.symbol;
    }

    return s;
}

```

---

## 214 Partition to K Equal Sum Subsets

Given an array of integers  $\text{nums}$  and a positive integer  $k$ , find whether it's possible to divide this array into  $k$  non-empty subsets whose sums are all equal.

Example 1:

Input:  $\text{nums} = [4, 3, 2, 3, 5, 2, 1]$ ,  $k = 4$  Output: True Explanation: It's possible to divide it into 4 subsets (5), (1, 4), (2,3), (2,3) with equal sums.

### 214.1 Java Solution

The easiest solution to this problem is DFS. We try to place each element to one of the bucket. The following is a Java solution and there is a diagram to show the execution of the helper() method using the given example. Note the improvement in the for loop.

```
public boolean canPartitionKSubsets(int[] nums, int k) {
    int sum = 0;
    for(int num: nums){
        sum+=num;
    }
    if(sum%k!=0){
        return false;
    }

    int share = sum/k;

    //sort array
    Arrays.sort(nums);

    int j=nums.length-1;
    if(nums[j]>share){
        return false;
    }
    while(j>=0 && nums[j]==share){
        j--;
        k--;
    }

    int[] buckets = new int[k];
    return helper(j, nums, share, buckets);
}

//put jth number to each bucket and recursively search
public boolean helper(int j, int[] nums, int share, int[] buckets){
    if(j<0){
        return true;
    }

    for(int i=0; i<buckets.length; i++){
        if(buckets[i]+nums[j]<=share){
            buckets[i]+=nums[j];
            if(helper(j-1, nums, share, buckets)){
                return true;
            }
            buckets[i]-=nums[j];
        }
    }
    return false;
}
```

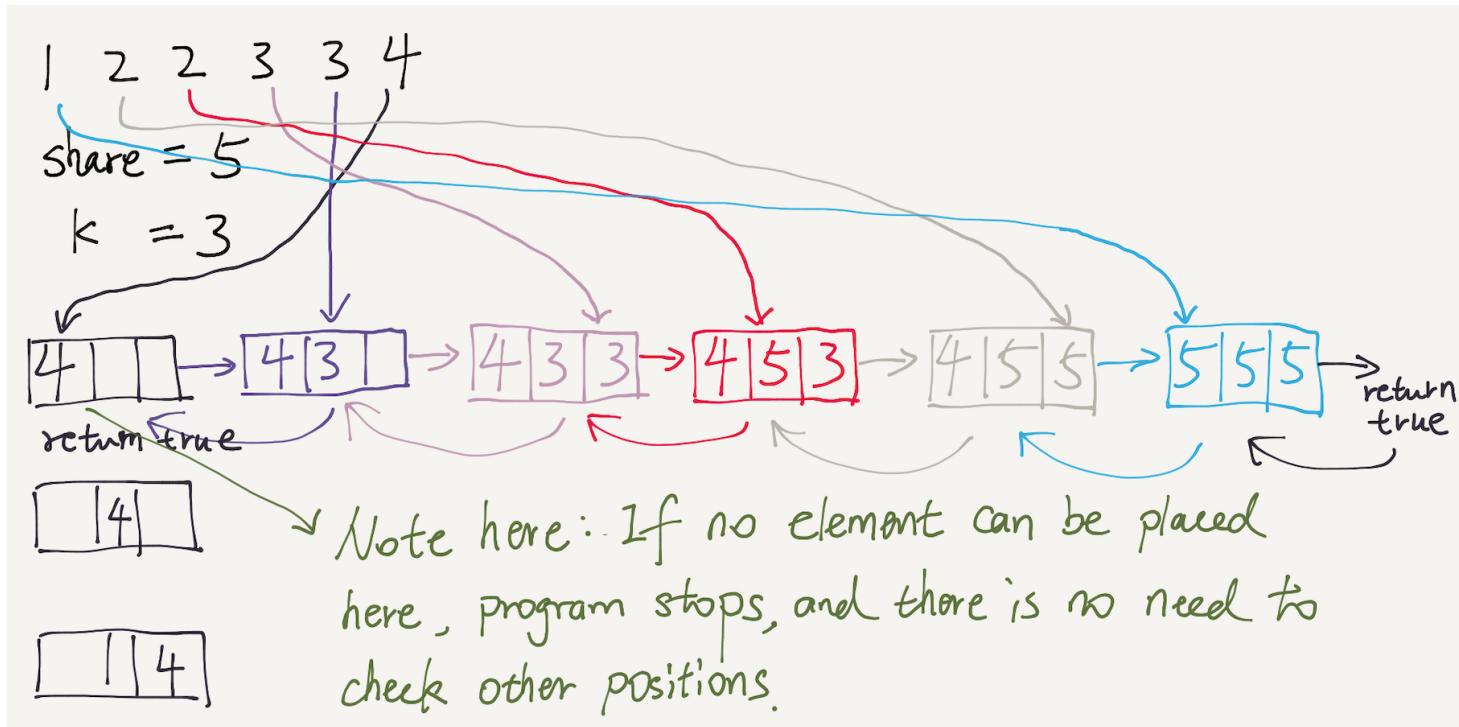
```

        return true;
    }
    buckets[i]-=nums[j];
}

if(buckets[i]==0) break; // if all elements are placed in a subset
}

return false;
}

```



# 215 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].

---

## 215.1 Java Solution 1 - Iteration

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(); j++) {
                // + add num[i] to different locations
                l.add(j, num[i]);

                ArrayList<Integer> temp = new ArrayList<Integer>(l);
                current.add(temp);

                //System.out.println(temp);

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

---

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

return result;
}
```

---

## 215.2 Java Solution 2 - Recursion

We can also recursively solve this problem. Swap each element with each element after it.

```
public List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    helper(0, nums, result);
    return result;
}

private void helper(int start, int[] nums, List<List<Integer>> result){
    if(start==nums.length-1){
        ArrayList<Integer> list = new ArrayList<>();
        for(int num: nums){
            list.add(num);
        }
        result.add(list);
        return;
    }

    for(int i=start; i<nums.length; i++){
        swap(nums, i, start);
        helper(start+1, nums, result);
        swap(nums, i, start);
    }
}

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

---

start=0

1 2 3

start=1

1 2 3  
| 3 2

start=2

1 2 3  
1 3 2

2 1 3

2 1 3  
2 3 12 1 3  
2 3 1

3 2 1

3 2 1  
3 1 23 2 1  
3 1 2

Since  $C(n) = 1 + C(n-1)$ , if we expand it, we can get time complexity is  $O(N!)$ .

# 216 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].

---

## 216.1 Java Solution 1

Based on [Permutation](#), we can add a set to track if an element is duplicate and no need to swap.

```
public List<List<Integer>> permuteUnique(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    helper(0, nums, result);
    return result;
}

private void helper(int start, int[] nums, List<List<Integer>> result){
    if(start==nums.length-1){
        ArrayList<Integer> list = new ArrayList<>();
        for(int num: nums){
            list.add(num);
        }
        result.add(list);
        return;
    }

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

    for(int i=start; i<nums.length; i++){
        if(set.contains(nums[i])){
            continue;
        }
        set.add(nums[i]);

        swap(nums, i, start);
        helper(start+1, nums, result);
        swap(nums, i, start);
    }
}

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

---

## 216.2 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!

# 217 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.)

## 217.1 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();
}
}
```

---

## 217.2 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();
    }
}
```

---

## 218 Number of Squareful Arrays

Given an array A of non-negative integers, the array is squareful if for every pair of adjacent elements, their sum is a perfect square.

Return the number of permutations of A that are squareful. Two permutations A<sub>1</sub> and A<sub>2</sub> differ if and only if there is some index i such that A<sub>1</sub>[i] != A<sub>2</sub>[i].

Example 1:

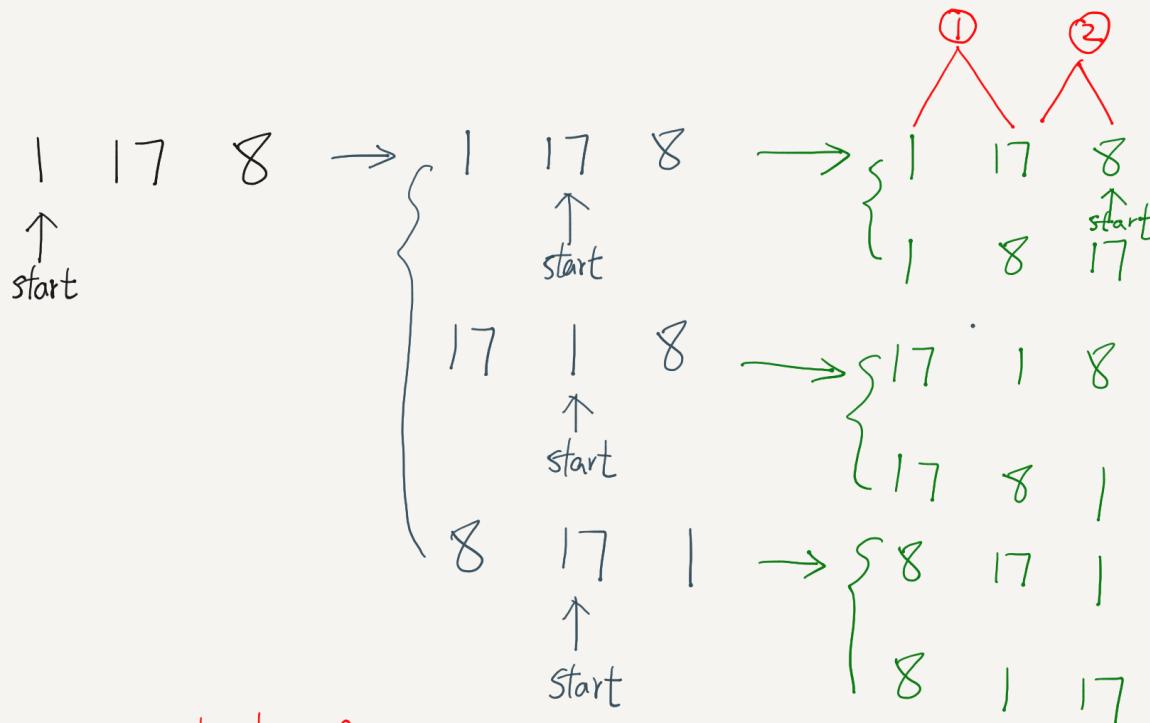
Input: [1,17,8] Output: 2 Explanation: [1,8,17] and [17,8,1] are the valid permutations.

Example 2:

Input: [2,2,2] Output: 1

### 218.1 Java Solution

This problem can be solved by using the similar method of [Permutation II](#). A set can be used to track if same element is swapped. The major difference is that we need to check if previous adjacent two neighbors are squareful.



- ① check if its previous two neighbors are squareful.  
     because elements after start do not swap
- ② If start is the last element, it does not swap, so  
     need to check if previous element and itself are squareful.

```

class Solution {
    int count = 0;

    public int numSquarefulPerms(int[] A) {
        Arrays.sort(A);
        helper(A, 0);
        return count;
    }

    void helper(int[] A, int start){
        //the adjacent two numbers before start
        if(start>1 && (!isSquareful(A[start], A[start-1]) || !isSquareful(A[start-1], A[start-2]))){
            return;
        }
        //if start is the last, then check start with its adjacent neighbor
        if(start==A.length-1 && !isSquareful(A[start], A[start-1])){
            return;
        }
    }
}

```

```
if(start==A.length-1){
    count++;
    return;
}

HashSet<Integer> set = new HashSet<>();
for(int i=start; i<A.length; i++){
    //use set to track if the same element is used
    if(set.contains(A[i])){
        continue;
    }
    set.add(A[i]);

    swap(A, i, start);
    helper(A, start+1);
    swap(A, i, start);
}
}

void swap(int[] A, int i, int j){
    int t = A[i];
    A[i] = A[j];
    A[j] = t;
}

boolean isSquareful(int a, int b){
    double root = Math.sqrt(a+b);
    return (root-Math.floor(root))==0;
}
}
```

---

# 219 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:

---

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

---

## 219.1 Java Solution 1 - DFS

This solution is simple and clear. In the dfs() method, left stands for the remaining number of (, right stands for the remaining number of ).

---

```
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);
    }
}
```

---

## 219.2 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("");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < result.size(); j++) {
            String str = result.get(j);
            if (str.length() < n * 2) {
                if (diff.get(j) == 0) {
                    result.add(str + "(");
                    diff.add(1);
                } else if (diff.get(j) == 1) {
                    result.add(str + ")");
                    diff.add(0);
                }
            }
        }
    }
}
```

---

```
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;
}
```

---

## 220 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:

---

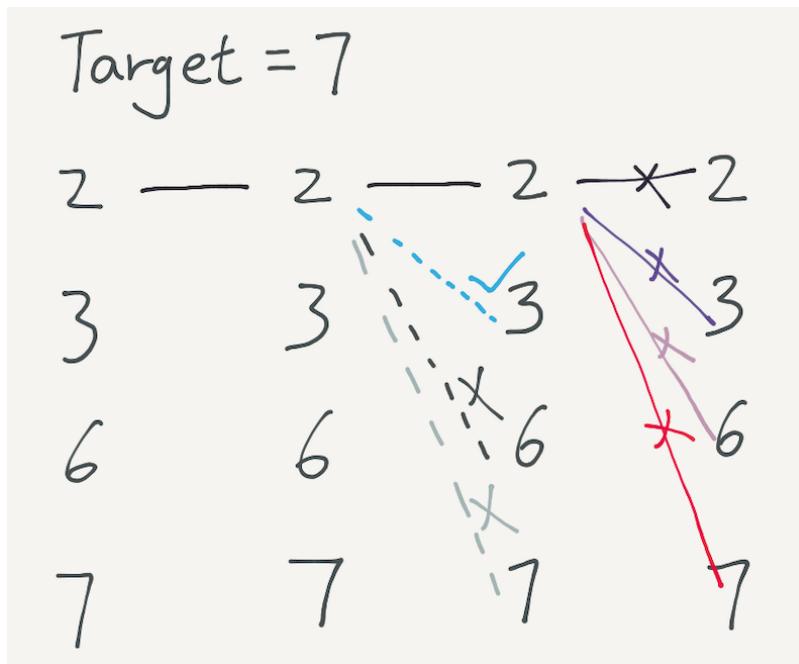
```
[7]
[2, 2, 3]
```

---

### 220.1 Java Solution

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

The following example shows how DFS works:



---

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> result = new ArrayList<>();
    List<Integer> temp = new ArrayList<>();
    helper(candidates, 0, target, 0, temp, result);
    return result;
}

private void helper(int[] candidates, int start, int target, int sum,
```

---

```
List<Integer> list, List<List<Integer>> result){  
    if(sum>target){  
        return;  
    }  
  
    if(sum==target){  
        result.add(new ArrayList<>(list));  
        return;  
    }  
  
    for(int i=start; i<candidates.length; i++){  
        list.add(candidates[i]);  
        helper(candidates, i, target, sum+candidates[i], list, result);  
        list.remove(list.size()-1);  
    }  
}
```

---

# 221 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.

## 221.1 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<List<Integer>> combinationSum2(int[] candidates, int target) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    List<Integer> curr = new ArrayList<Integer>();
    Arrays.sort(candidates);
    helper(result, curr, 0, target, candidates);
    return result;
}

public void helper(List<List<Integer>> result, List<Integer> curr, int start, int target, int[]
    candidates){
    if(target==0){
        result.add(new ArrayList<Integer>(curr));
        return;
    }
    if(target<0){
        return;
    }

    int prev=-1;
    for(int i=start; i<candidates.length; i++){
        if(prev!=candidates[i]){ // each time start from different element
            curr.add(candidates[i]);
            helper(result, curr, i+1, target-candidates[i], candidates); // and use next element only
            curr.remove(curr.size()-1);
            prev=candidates[i];
        }
    }
}
```

---

## 222 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:  $[[1,2,6], [1,3,5], [2,3,4]]$

### 222.1 Analysis

Related problems: [Combination Sum](#), [Combination Sum II](#).

### 222.2 Java Solution

---

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

public void helper(List<List<Integer>> result, List<Integer> curr, int k, int start, int sum){
    if(sum<0){
        return;
    }

    if(sum==0 && curr.size()==k){
        result.add(new ArrayList<Integer>(curr));
        return;
    }

    for(int i=start; i<=9; i++){
        curr.add(i);
        helper(result, curr, k, i+1, sum-i);
        curr.remove(curr.size()-1);
    }
}
```

---

## 223 Combination Sum IV

Given an integer array with all positive numbers and no duplicates, find the number of possible combinations that add up to a positive integer target.

### 223.1 Java Solution

This problem is similar to [Coin Change](#). It's a typical dynamic programming problem.

---

```
public int combinationSum4(int[] nums, int target) {  
    if(nums==null || nums.length==0)  
        return 0;  
  
    int[] dp = new int[target+1];  
  
    dp[0]=1;  
  
    for(int i=0; i<=target; i++){  
        for(int num: nums){  
            if(i+num<=target){  
                dp[i+num]+=dp[i];  
            }  
        }  
    }  
  
    return dp[target];  
}
```

---

# 224 Wildcard Matching

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

## 224.1 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();
}
```

---

# 225 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

## 225.1 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.

## 225.2 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));

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

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

```

        }
    }
}
```

---

### 225.3 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;
}
```

---

# 226 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  $(1+2)*(3+4)=21$

## 226.1 Analysis

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

## 226.2 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));
        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 num1 : result1) {
            for (int num2 : result2) {
                int sum = num1 + num2;
                if (sum == target) {
                    result.add(sum);
                }
            }
        }
    }
}
```

```
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;  
}
```

---

## 227 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.

### 227.1 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;
}
```

---

## 228 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 "+-+".

### 228.1 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;
}
```

---

### 228.2 Time Complexity

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

## 229 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.

### 229.1 Java Solution

---

```
public boolean wordPattern(String pattern, String str) {
    String[] arr = str.split(" ");

    //prevent out of boundary problem
    if(arr.length != pattern.length())
        return false;

    HashMap<Character, String> map = new HashMap<Character, String>();
    for(int i=0; i<pattern.length(); i++){
        char c = pattern.charAt(i);
        if(map.containsKey(c)){
            String value = map.get(c);
            if(!value.equals(arr[i])){
                return false;
            }
        }else if (map.containsValue(arr[i])){
            return false;
        }
        map.put(c, arr[i]);
    }

    return true;
}
```

---

## 230 Word Pattern II

This is the extension problem of [Word Pattern I](#).

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 substring in str.

Examples: pattern = "abab", str = "redblueredblue" should return true. pattern = "aaaa", str = "asdasdasdasd" should return true. pattern = "aabb", str = "xyzabcxzyabc" should return false.

### 230.1 Java Solution

```
public boolean wordPatternMatch(String pattern, String str) {  
    if(pattern.length()==0 && str.length()==0)  
        return true;  
    if(pattern.length()==0)  
        return false;  
  
    HashMap<Character, String> map = new HashMap<Character, String>();  
  
    return helper(pattern, str, 0, 0, map);  
}  
  
public boolean helper(String pattern, String str, int i, int j, HashMap<Character, String> map){  
    if(i==pattern.length() && j==str.length())  
        return true;  
    if(i>=pattern.length() || j>=str.length())  
        return false;  
  
    char c = pattern.charAt(i);  
    for(int k=j+1; k<=str.length(); k++){  
        String sub = str.substring(j, k);  
        if(!map.containsKey(c) && !map.containsValue(sub)){  
            map.put(c, sub);  
            if(helper(pattern, str, i+1, k, map))  
                return true;  
            map.remove(c);  
        } else if(map.containsKey(c) && map.get(c).equals(sub)){  
            if(helper(pattern, str, i+1, k, map))  
                return true;  
        }  
    }  
  
    return false;  
}
```

Since containsValue() method is used here, the time complexity is  $O(n)$ . We can use another set to track the value set which leads to time complexity of  $O(1)$ :

```

public boolean wordPatternMatch(String pattern, String str) {
    if(pattern.length()==0 && str.length()==0)
        return true;
    if(pattern.length()==0)
        return false;

    HashMap<Character, String> map = new HashMap<Character, String>();
    HashSet<String> set = new HashSet<String>();
    return helper(pattern, str, 0, 0, map, set);
}

public boolean helper(String pattern, String str, int i, int j, HashMap<Character, String> map,
    HashSet<String> set){
    if(i==pattern.length() && j==str.length()){
        return true;
    }

    if(i>=pattern.length() || j>=str.length())
        return false;

    char c = pattern.charAt(i);
    for(int k=j+1; k<=str.length(); k++){
        String sub = str.substring(j, k);
        if(!map.containsKey(c) && !set.contains(sub)){
            map.put(c, sub);
            set.add(sub);
            if(helper(pattern, str, i+1, k, map, set))
                return true;
            map.remove(c);
            set.remove(sub);
        }else if(map.containsKey(c) && map.get(c).equals(sub)){
            if(helper(pattern, str, i+1, k, map, set))
                return true;
        }
    }
    return false;
}

```

---

The time complexity then is  $f(n) = n * (n-1) * \dots * 1 = n!$ .

# 231 Scramble String

Given two strings s<sub>1</sub> and s<sub>2</sub> of the same length, determine if s<sub>2</sub> is a scrambled string of s<sub>1</sub>.

## 231.1 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;
}
```

---

## 232 Remove Invalid Parentheses

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses ( and ).

Examples: "())()" ->["()", "()"] "(a)()" ->"(a)", "(a)" ] ")" ->[""]

### 232.1 Java Solution

This problem can be solve by using DFS.

```
public class Solution {
    ArrayList<String> result = new ArrayList<String>();
    int max=0;

    public List<String> removeInvalidParentheses(String s) {
        if(s==null)
            return result;

        dfs(s, "", 0, 0);
        if(result.size()==0){
            result.add("");
        }

        return result;
    }

    public void dfs(String left, String right, int countLeft, int maxLeft){
        if(left.length()==0){
            if(countLeft==0 && right.length()!=0){
                if(maxLeft > max){
                    max = maxLeft;
                }

                if(maxLeft==max && !result.contains(right)){
                    result.add(right);
                }
            }
        }

        return;
    }

    if(left.charAt(0)=='('){
        dfs(left.substring(1), right+"(", countLeft+1, maxLeft+1); //keep (
        dfs(left.substring(1), right, countLeft, maxLeft); //drop (
    }else if(left.charAt(0)==')'){
        if(countLeft>0){
            dfs(left.substring(1), right+")", countLeft-1, maxLeft);
        }
    }

    dfs(left.substring(1), right, countLeft, maxLeft);
}
```

```
    }else{
        dfs(left.substring(1), right+String.valueOf(left.charAt(0)), countLeft, maxLeft);
    }
}
```

---

## 233 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".

### 233.1 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;
}
```

---

### 233.2 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;

    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();
}
```

---

## 234 Lexicographical Numbers

Given an integer n, return 1 - n in lexicographical order.

For example, given 13, return: [1,10,11,12,13,2,3,4,5,6,7,8,9].

Please optimize your algorithm to use less time and space. The input size may be as large as 5,000,000.

### 234.1 Java Solution - DFS

```
public List<Integer> lexicalOrder(int n) {
    int c=0;
    int t=n;
    while(t>0){
        c++;
        t=t/10;
    }

    ArrayList<Integer> result = new ArrayList<Integer>();
    char[] num = new char[c];

    helper(num, 0, n, result);

    return result;
}

public void helper(char[] num, int i, int max, ArrayList<Integer> result){
    if(i==num.length){
        int val = convert(num);
        if(val <=max)
            result.add(val);
        return;
    }

    if(i==0){
        for(char c='1'; c<='9'; c++){
            num[i]=c;
            helper(num, i+1, max, result);
        }
    }else{
        num[i]='a';
        helper(num, num.length, max, result);

        for(char c='0'; c<='9'; c++){
            num[i]=c;
            helper(num, i+1, max, result);
        }
    }
}

private int convert(char[] arr){
```

---

```

int result=0;
for(int i=0; i<arr.length; i++){
    if(arr[i]>='0'&&arr[i]<='9')
        result = result*10+arr[i]-'0';
    else
        break;
}
return result;
}

```

---

## 234.2 Java Solution 2 - Comparator

---

```

public List<Integer> lexicalOrder(int n) {
    List<String> list = new ArrayList<>();
    for(int i=1;i<=n;i++){
        list.add(String.valueOf(i));
    }

    Collections.sort(list, new Comparator<String>(){
        public int compare(String a, String b){
            int i=0;
            while(i<a.length()&&i<b.length()){
                if(a.charAt(i)!=b.charAt(i)){
                    return a.charAt(i)-b.charAt(i);
                }
                i++;
            }

            if(i>=a.length()){
                return -1;
            }

            return 1;
        }
    });
}

List<Integer> result = new ArrayList<>();
for(String s: list){
    result.add(Integer.parseInt(s));
}

return result;
}

```

---

## 235 Combinations

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],  
 ]
```

### 235.1 Java Solution

```
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);  
     }  
 }
```

# 236 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"].

## 236.1 Java Solution 1 - DFS

This problem can be solved by a typical DFS algorithm. DFS problems are very similar and can be solved by using a simple recursion.

```
public List<String> letterCombinations(String digits) {
    HashMap<Character, char[]> dict = new HashMap<Character, char[]>();
    dict.put('2',new char[]{'a','b','c'});
    dict.put('3',new char[]{'d','e','f'});
    dict.put('4',new char[]{'g','h','i'});
    dict.put('5',new char[]{'j','k','l'});
    dict.put('6',new char[]{'m','n','o'});
    dict.put('7',new char[]{'p','q','r','s'});
    dict.put('8',new char[]{'t','u','v'});
    dict.put('9',new char[]{'w','x','y','z'});

    List<String> result = new ArrayList<String>();
    if(digits==null||digits.length()==0){
        return result;
    }

    char[] arr = new char[digits.length()];
    helper(digits, 0, dict, result, arr);

    return result;
}

private void helper(String digits, int index, HashMap<Character, char[]> dict,
                   List<String> result, char[] arr){
    if(index==digits.length()){
        result.add(new String(arr));
        return;
    }

    char number = digits.charAt(index);
    char[] candidates = dict.get(number);
    for(int i=0; i<candidates.length; i++){
        arr[index]=candidates[i];
        helper(digits, index+1, dict, result, arr);
    }
}
```

Time complexity is  $O(k^n)$ , where  $k$  is the biggest number of letters a digit can map ( $k=4$ ) and  $n$  is the length of the digit string.

## 236.2 Java Solution 2 - BFS

---

```
public List<String> letterCombinations(String digits) {
    String[] dict = new String[]{"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};
    LinkedList<StringBuilder> queue = new LinkedList<>();

    for(int i=0; i<digits.length(); i++){
        int d = digits.charAt(i) - '0';
        String option = dict[d];

        if(i==0){
            for(int j=0; j<option.length(); j++){
                queue.offer(new StringBuilder().append(option.charAt(j)));
            }
        }else{
            LinkedList<StringBuilder> temp = new LinkedList<>();
            while(!queue.isEmpty()){
                StringBuilder sb = queue.poll();
                for(int j=0; j<option.length(); j++){
                    temp.offer(new StringBuilder(sb).append(option.charAt(j)));
                }
            }
            queue.addAll(temp);
        }
    }

    List<String> result = new ArrayList<>();
    while(!queue.isEmpty()){
        result.add(queue.poll().toString());
    }

    return result;
}
```

---

# 237 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"].

## 237.1 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);
        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);

}
}
```

---

## 238 Factor Combinations

Numbers can be regarded as product of its factors. For example,

---

```
8 = 2 × 2 × 2;  
= 2 × 4.
```

---

Write a function that takes an integer n and return all possible combinations of its factors.

Note: You may assume that n is always positive. Factors should be greater than 1 and less than n.

### 238.1 Java Solution

---

```
public List<List<Integer>> getFactors(int n) {  
    List<List<Integer>> result = new ArrayList<List<Integer>>();  
    List<Integer> list = new ArrayList<Integer>();  
    helper(2, 1, n, result, list);  
    return result;  
}  
  
public void helper(int start, int product, int n, List<List<Integer>> result, List<Integer> curr){  
    if(start>n || product > n )  
        return ;  
  
    if(product==n) {  
        ArrayList<Integer> t = new ArrayList<Integer>(curr);  
        result.add(t);  
        return;  
    }  
  
    for(int i=start; i<n; i++){  
        if(i*product>n)  
            break;  
  
        if(n%i==0){  
            curr.add(i);  
            helper(i, i*product, n, result, curr);  
            curr.remove(curr.size()-1);  
        }  
    }  
}
```

---

# 239 Subsets

Given a set of distinct integers,  $S$ , return all possible subsets.

Note: 1) Elements in a subset must be in non-descending order. 2) The solution set must not contain duplicate subsets.

---

For example, given  $S = [1,2,3]$ , the method returns:

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

---

## 239.1 Thoughts

Given a set  $S$  of  $n$  distinct integers, there is a relation between  $S_n$  and  $S_{n-1}$ . The subset of  $S_{n-1}$  is the union of subset of  $S_{n-1}$  and each element in  $S_{n-1} +$  one more element. Therefore, a Java solution can be quickly formalized.

## 239.2 Java Solution

```
public ArrayList<ArrayList<Integer>> subsets(int[] S) {  
    if (S == null)  
        return null;  
  
    Arrays.sort(S);  
  
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();  
  
    for (int i = 0; i < S.length; i++) {  
        ArrayList<ArrayList<Integer>> temp = new ArrayList<ArrayList<Integer>>();  
  
        //get sets that are already in result  
        for (ArrayList<Integer> a : result) {  
            temp.add(new ArrayList<Integer>(a));  
        }  
  
        //add S[i] to existing sets  
        for (ArrayList<Integer> a : temp) {  
            a.add(S[i]);  
        }  
  
        //add S[i] only as a set  
        ArrayList<Integer> single = new ArrayList<Integer>();  
        single.add(S[i]);  
        temp.add(single);  
    }  
    result.addAll(temp);  
}
```

```
single.add(S[i]);
temp.add(single);

result.addAll(temp);
}

//add empty set
result.add(new ArrayList<Integer>());

return result;
}
```

---

## 240 Subsets II

Given a set of distinct integers, S, return all possible subsets.

Note: Elements in a subset must be in non-descending order. The solution set must not contain duplicate subsets. For example, If S = [1,2,3], a solution is:

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

### 240.1 Thoughts

Comparing this problem with [Subsets](#) can help better understand the problem.

### 240.2 Java Solution

```
public ArrayList<ArrayList<Integer>> subsetsWithDup(int[] num) {  
    if (num == null)  
        return null;  
  
    Arrays.sort(num);  
  
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();  
    ArrayList<ArrayList<Integer>> prev = new ArrayList<ArrayList<Integer>>();  
  
    for (int i = num.length-1; i >= 0; i--) {  
  
        //get existing sets  
        if (i == num.length - 1 || num[i] != num[i + 1] || prev.size() == 0) {  
            prev = new ArrayList<ArrayList<Integer>>();  
            for (int j = 0; j < result.size(); j++) {  
                prev.add(new ArrayList<Integer>(result.get(j)));  
            }  
        }  
  
        //add current number to each element of the set  
        for (ArrayList<Integer> temp : prev) {  
            temp.add(0, num[i]);  
        }  
  
        //add each single number as a set, only if current element is different with previous  
    }
```

```
if (i == num.length - 1 || num[i] != num[i + 1]) {
    ArrayList<Integer> temp = new ArrayList<Integer>();
    temp.add(num[i]);
    prev.add(temp);
}

//add all set created in this iteration
for (ArrayList<Integer> temp : prev) {
    result.add(new ArrayList<Integer>(temp));
}
}

//add empty set
result.add(new ArrayList<Integer>());

return result;
}
```

---

Feed the method [1,2,3] the following will be result at each iteration.

[2]  
[2][2,2]  
[2][2,2][1,2][1,2,2][1]  
Get [] finally.

---

## 241 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.

### 241.1 Java Solution 1 - Dynamic Programming (Looking Backward)

Given an amount of 6 and coins [1,2,5], we can look backward in the dp array.

$i$	0	1	2	3	4	5	6
$dp$	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
			$\nearrow_{i-2}$	$\nwarrow_{i-1}$			
			$i = i$	$2 = i$	$dp[i-1] = 1$	$dp[i-2] = 1$	$dp[i-5] \times$ $\downarrow$ $1+1=2$
		$\downarrow$	$\downarrow$	$\downarrow$	$dp[i-1] = 1$	$dp[i-2] = 1$	
		$\text{set to } 1$	$\text{set to } 1$				
		$\textcircled{1}$	$\textcircled{2}$				

---

Let  $dp[i]$  to be the minimum number of coins required to get the amount  $i$ .

$dp[i] = 1$ , if  $i == \text{coin}$

otherwise,  $dp[i] = \min(dp[i-\text{coin}]+1, dp[i])$  if  $dp[i-\text{coin}]$  is reachable.

We initially set  $dp[i]$  to be MAX\_VALUE.

---

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

    int[] dp = new int[amount+1];

    Arrays.fill(dp, Integer.MAX_VALUE);
    dp[0]=0;

    for(int i=1; i<=amount; i++){
        for(int coin: coins){
            if(i==coin){
                dp[i]=1;
            }else if(i>coin){
                if(dp[i-coin]==Integer.MAX_VALUE){
```

```

        continue;
    }
    dp[i]=Math.min(dp[i-coin]+1, dp[i]);
}
}

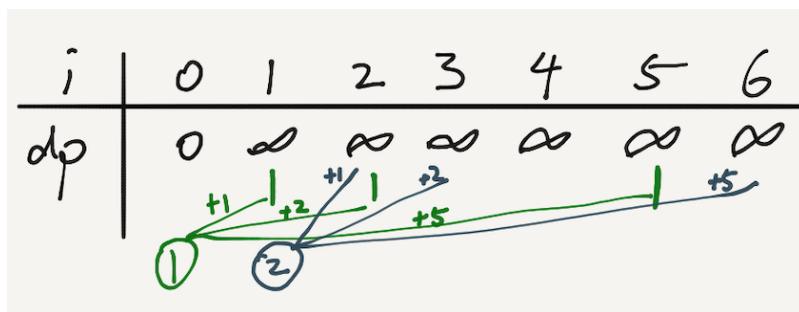
if(dp[amount]==Integer.MAX_VALUE){
    return -1;
}

return dp[amount];
}

```

Time complexity is  $O(\text{amount} * \text{num\_of\_coins})$  and space complexity is  $O(\text{amount})$ .

## 241.2 Java Solution 2 - Dynamic Programming (Looking Forward)



Let  $dp[i]$  to be the minimum number of coins required to get the amount  $i$ .  
 $dp[i+coin] = \min(dp[i+coin], dp[i]+1)$  if  $dp[i]$  is reachable.  
 $dp[i+coin] = dp[i+coin]$  if  $dp[i]$  is not reachable.  
We initially set  $dp[i]$  to be  $\text{MAX\_VALUE}$ .

Here is the Java code:

```

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

    int[] dp = new int[amount+1];

    Arrays.fill(dp, Integer.MAX_VALUE);
    dp[0]=0;

    for(int i=0; i<=amount; i++){
        if(dp[i]==Integer.MAX_VALUE){
            continue;
        }

        for(int coin: coins){
            if(i<=amount-coin){ //handle case when coin is Integer.MAX_VALUE
                dp[i+coin] = Math.min(dp[i]+1, dp[i+coin]);
            }
        }
    }
}

```

---

```

        }
    }

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

    return dp[amount];
}

```

---

Time and space are the same as Solution 1.

### 241.3 Java Solution 3 - Breath First Search (BFS)

Dynamic programming problems can often be solved by using BFS.

We can view this problem as going to a target position with steps that are allowed 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;
}

```

---

# 242 Palindrome Partitioning

## 242.1 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"]  
]
```

## 242.2 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);  
        }  
    }  
}  
  
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;
}

```

---

## 242.3 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;
}

```

---

# 243 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.

## 243.1 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.

## 243.2 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];
}
```

---

## 244 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.

### 244.1 Java Solution 1 - Dynamic Programming

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

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

    if(nums.length==1)
        return nums[0];

    int[] dp = new int[nums.length];
    dp[0]=nums[0];
    dp[1]=Math.max(nums[0], nums[1]);

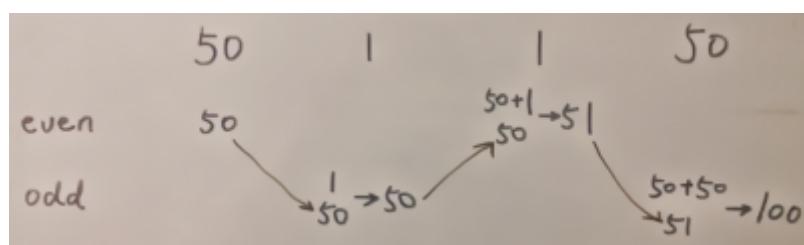
    for(int i=2; i<nums.length; i++){
        dp[i] = Math.max(dp[i-2]+nums[i], dp[i-1]);
    }

    return dp[nums.length-1];
}
```

### 244.2 Java Solution 2

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

50 1 1 50



---

```

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

    int even = 0;
    int odd = 0;

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

    return even > odd ? even : odd;
}

```

---

### 244.3 Java Solution 3 - Dynamic Programming with Memorization

---

```

public int rob(int[] nums) {
    if(nums.length==0){
        return 0;
    }

    int[] mem = new int[nums.length+1];
    Arrays.fill(mem, -1);

    mem[0] = 0;

    return helper(nums.length, mem, nums);
}

private int helper(int size, int[] mem, int[] nums){
    if(size <1){
        return 0;
    }

    if(mem[size]!=-1){
        return mem[size];
    }

    //two cases
    int firstSelected = helper(size-2, mem, nums) + nums[nums.length -size];
    int firstUnselected = helper(size-1, mem, nums);

    return mem[size] = Math.max(firstSelected, firstUnselected);
}

```

---

# 245 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.

## 245.1 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.

## 245.2 Java Solution

---

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

    if(nums.length==1)
        return nums[0];

    int max1 = robHelper(nums, 0, nums.length-2);
    int max2 = robHelper(nums, 1, nums.length-1);

    return Math.max(max1, max2);
}

public int robHelper(int[] nums, int i, int j){
    if(i==j){
        return nums[i];
    }

    int[] dp = new int[nums.length];
    dp[i]=nums[i];
    dp[i+1]=Math.max(nums[i+1], dp[i]);

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

    return dp[j];
}
```

---

# 246 House Robber III

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

## 246.1 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.

## 246.2 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;
}
```

---

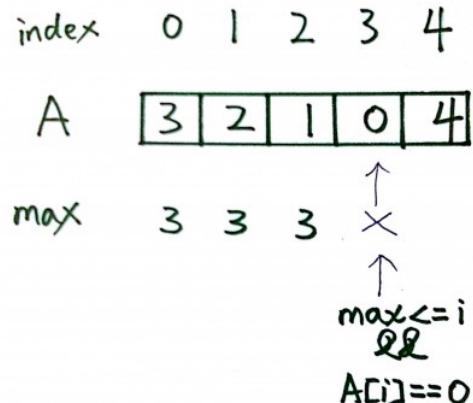
# 247 Jump Game

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.

## 247.1 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:  $i + A[i]$ .



## 247.2 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  
        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;
}
```

---

# 248 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.)

## 248.1 Analysis

This is an extension of [Jump Game](#).

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

## 248.2 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;
}
```

---

# 249 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.

## 249.1 Java Solution

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

---

```
public int maxProfit(int[] prices) {
    if(prices==null||prices.length<=1)
        return 0;

    int min=prices[0]; // min so far
    int result=0;

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

    return result;
}
```

---

# 250 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).

## 250.1 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.

## 250.2 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;
}
```

---

# 251 Best Time to Buy and Sell Stock III

Say you have an array for which the  $i^{\text{th}}$  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).

## 251.1 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

## 251.2 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
    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;  
}
```

---

# 252 Best Time to Buy and Sell Stock IV

## 252.1 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).

## 252.2 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.

## 252.3 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];
}
```

---

## 252.4 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];
}
```

---

# 253 Dungeon Game

Example:

---

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

---

## 253.1 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];
}
```

---

## 254 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.

### 254.1 Java Solution

This problem can be solved by using dynamic programming. It is similar to the problem of counting ways of climbing stairs. The relation is  $dp[n]=dp[n-1]+dp[n-2]$ .

---

```
public int numDecodings(String s) {
    if(s==null || s.length()==0 || s.charAt(0)=='0')
        return 0;
    if(s.length()==1)
        return 1;

    int[] dp = new int[s.length()];
    dp[0]=1;
    if(Integer.parseInt(s.substring(0,2))>26){
        if(s.charAt(1)!='0'){
            dp[1]=1;
        }else{
            dp[1]=0;
        }
    }else{
        if(s.charAt(1)!='0'){
            dp[1]=2;
        }else{
            dp[1]=1;
        }
    }

    for(int i=2; i<s.length(); i++){
        if(s.charAt(i)!='0'){
            dp[i]+=dp[i-1];
        }

        int val = Integer.parseInt(s.substring(i-1, i+1));
        if(val<=26 && val >=10){
            dp[i]+=dp[i-2];
        }
    }

    return dp[s.length()-1];
}
```

---

## 255 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  $13 = 4 + 9$ .

### 255.1 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];
}
```

---

# 256 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".

## 256.1 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.

## 256.2 Dynamic Programming

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

- Define an array  $t[]$  such that  $t[i]==\text{true} \Rightarrow 0-(i-1)$  can be segmented using dictionary
- Initial state  $t[0] == \text{true}$

---

```
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:  $O(\text{string length} * \text{dict size})$ .

### 256.3 Java Solution 3 - Simple and Efficient

In Solution 2, if the size of the dictionary is very large, the time is bad. Instead we can solve the problem in  $O(n^2)$  time ( $n$  is the length of the string).

---

```

public boolean wordBreak(String s, Set<String> wordDict) {
    int[] pos = new int[s.length()+1];

    Arrays.fill(pos, -1);

    pos[0]=0;

    for(int i=0; i<s.length(); i++){
        if(pos[i]!=-1){
            for(int j=i+1; j<=s.length(); j++){
                String sub = s.substring(i, j);
                if(wordDict.contains(sub)){
                    pos[j]=i;
                }
            }
        }
    }

    return pos[s.length()]!= -1;
}

```

---

## 256.4 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](#).

## 257 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"].

### 257.1 Java Solution 1 - 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++){
        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);
        }
    }
}

```

## 257.2 Java Solution 2 - Simplified

```

public List<String> wordBreak(String s, Set<String> wordDict) {
    ArrayList<String> [] pos = new ArrayList[s.length()+1];
    pos[0]=new ArrayList<String>();

    for(int i=0; i<s.length(); i++){
        if(pos[i]!=null){
            for(int j=i+1; j<=s.length(); j++){
                String sub = s.substring(i,j);
                if(wordDict.contains(sub)){
                    if(pos[j]==null){

```

```

        ArrayList<String> list = new ArrayList<String>();
        list.add(sub);
        pos[j]=list;
    }else{
        pos[j].add(sub);
    }

}
}
}

if(pos[s.length()]==null){
    return new ArrayList<String>();
}else{
    ArrayList<String> result = new ArrayList<String>();
    dfs(pos, result, "", s.length());
    return result;
}
}

public void dfs(ArrayList<String> [] pos, ArrayList<String> result, String curr, int i){
    if(i==0){
        result.add(curr.trim());
        return;
    }

    for(String s: pos[i]){
        String combined = s + " " + curr;
        dfs(pos, result, combined, i-s.length());
    }
}

```

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".

# 258 Minimum Window Subsequence

Given strings S and T, find the minimum (contiguous) substring W of S, so that T is a subsequence of W.

If there is no such window in S that covers all characters in T, return the empty string "". If there are multiple such minimum-length windows, return the one with the left-most starting index.

Example 1:

Input: S = "abcdebdde", T = "bde" Output: "bcde" Explanation: "bcde" is the answer because it occurs before "bdde" which has the same length. "deb" is not a smaller window because the elements of T in the window must occur in order.

## 258.1 Java Solution 1 - Two pointers

In a brute-force way, this problem can be solved by using two pointers which iterate over characters of the two strings respectively.

---

```
public String minWindow(String S, String T) {
    int start=0;
    String result = "";

    while(start<S.length()){
        int j=0;

        for(int i=start; i<S.length(); i++){
            if(S.charAt(i)==T.charAt(j)&&j==0){
                start=i;
            }

            if(S.charAt(i)==T.charAt(j)){
                j++;
            }

            if(j==T.length()){
                if(result.equals("")||(i-start+1)<result.length()){
                    result = S.substring(start, i+1);
                }
                start=start+1;
                break;
            }
        }

        if(i==S.length()-1){
            return result;
        }
    }

    return result;
}
```

---

# 259 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.

## 259.1 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.

## 259.2 Java Solution

---

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

---

```
    return result*result;
}
```

---

# 260 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.

## 260.1 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;  
}
```

## 260.2 Java Solution 2: Dynamic Programming

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

---

# 261 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?

## 261.1 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;
}
```

---

## 261.2 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];
}

```

---

### 261.3 Java Solution 3 - Dynamic Programming with Memorization

---

```

public int uniquePaths(int m, int n) {
    int[][] mem = new int[m][n];

    //init with -1 value
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            mem[i][j]=-1;
        }
    }

    return helper(mem, m-1, n-1);
}

private int helper(int[][] mem, int m, int n){
    //edge has only one path
    if(m==0 || n==0){
        mem[m][n]=1;
        return 1;
    }

    if(mem[m][n]!=-1){
        return mem[m][n];
    }

    mem[m][n] = helper(mem, m, n-1) + helper(mem, m-1, n);

    return mem[m][n];
}

```

---

# 262 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.

## 262.1 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  
    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];
}
```

---

## 263 Paint House

There are a row of n houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a  $n \times 3$  cost matrix. For example,  $\text{costs}[0][0]$  is the cost of painting house 0 with color red;  $\text{costs}[1][2]$  is the cost of painting house 1 with color green, and so on... Find the minimum cost to paint all houses.

### 263.1 Java Solution

A typical DP problem.

---

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

    for(int i=1; i<costs.length; i++){
        costs[i][0] += Math.min(costs[i-1][1], costs[i-1][2]);
        costs[i][1] += Math.min(costs[i-1][0], costs[i-1][2]);
        costs[i][2] += Math.min(costs[i-1][0], costs[i-1][1]);
    }

    int m = costs.length-1;
    return Math.min(Math.min(costs[m][0], costs[m][1]), costs[m][2]);
}
```

---

Or a different way of writing the code without original array value changed.

---

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

    int[][] matrix = new int[3][costs.length];

    for(int j=0; j<costs.length; j++){
        if(j==0){
            matrix[0][j]=costs[j][0];
            matrix[1][j]=costs[j][1];
            matrix[2][j]=costs[j][2];
        }else{
            matrix[0][j]=Math.min(matrix[1][j-1], matrix[2][j-1])+costs[j][0];
            matrix[1][j]=Math.min(matrix[0][j-1], matrix[2][j-1])+costs[j][1];
            matrix[2][j]=Math.min(matrix[0][j-1], matrix[1][j-1])+costs[j][2];
        }
    }

    int result = Math.min(matrix[0][costs.length-1], matrix[1][costs.length-1]);
    result = Math.min(result, matrix[2][costs.length-1]);
}
```

```
    return result;  
}
```

---

## 264 Paint House II

There are a row of n houses, each house can be painted with one of the k colors. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a n x k cost matrix. For example, costs[0][0] is the cost of painting house 0 with color 0; costs[1][2] is the cost of painting house 1 with color 2, and so on... Find the minimum cost to paint all houses.

### 264.1 Java Solution

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

    int preMin=0;
    int preSecond=0;
    int preIndex=-1;

    for(int i=0; i<costs.length; i++){
        int currMin=Integer.MAX_VALUE;
        int currSecond = Integer.MAX_VALUE;
        int currIndex = 0;

        for(int j=0; j<costs[0].length; j++){
            if(preIndex==j){
                costs[i][j] += preSecond;
            }else{
                costs[i][j] += preMin;
            }

            if(currMin>costs[i][j]){
                currSecond = currMin;
                currMin=costs[i][j];
                currIndex = j;
            } else if(currSecond>costs[i][j] ){
                currSecond = costs[i][j];
            }
        }

        preMin=currMin;
        preSecond=currSecond;
        preIndex =currIndex;
    }

    int result = Integer.MAX_VALUE;
    for(int j=0; j<costs[0].length; j++){
        if(result>costs[costs.length-1][j]){
            result = costs[costs.length-1][j];
        }
    }
}
```

```
    }
    return result;
}
```

---

# 265 Edit Distance in Java

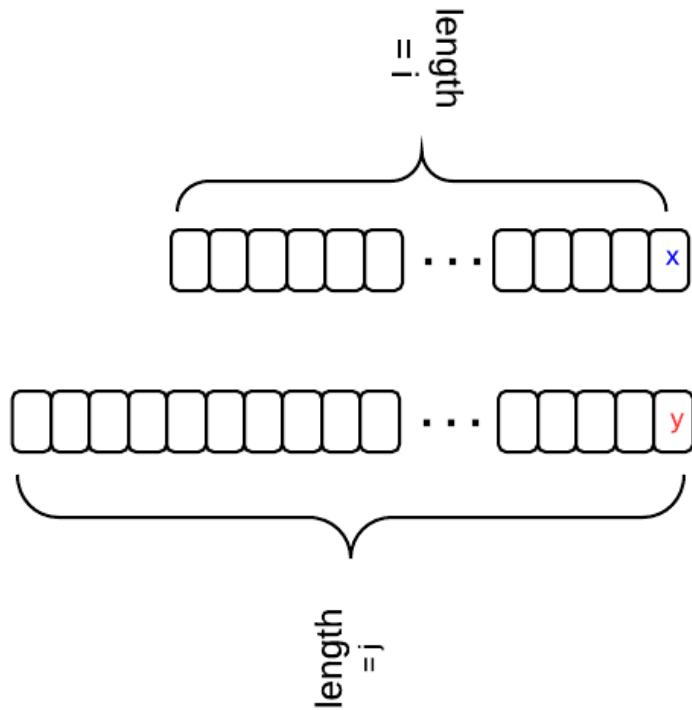
From Wiki:

*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.

## 265.1 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 it's last character is " $x$ "; the second string has length  $j$  and its last character is " $y$ ". The following diagram shows the relation.



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

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

## 265.2 Java Solution 1 - Iteration

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

---

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

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

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

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

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

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

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

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

---

## 265.3 Java Solution 2 - Recursion

We can write the solution in recursion.

---

```
public int minDistance(String word1, String word2) {
    int m=word1.length();
    int n=word2.length();
    int[][] mem = new int[m][n];
    for(int[] arr: mem){

```

```
        Arrays.fill(arr, -1);
    }
    return calDistance(word1, word2, mem, m-1, n-1);
}

private int calDistance(String word1, String word2, int[][] mem, int i, int j){
    if(i<0){
        return j+1;
    }else if(j<0){
        return i+1;
    }

    if(mem[i][j]!=-1){
        return mem[i][j];
    }

    if(word1.charAt(i)==word2.charAt(j)){
        mem[i][j]=calDistance(word1, word2, mem, i-1, j-1);
    }else{
        int prevMin = Math.min(calDistance(word1, word2, mem, i, j-1), calDistance(word1, word2, mem, i-1, j));
        prevMin = Math.min(prevMin, calDistance(word1, word2, mem, i-1, j-1));
        mem[i][j]=1+prevMin;
    }

    return mem[i][j];
}
```

# 266 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.

## 266.1 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.

## 266.2 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 numDistinct(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()];  
}
```

## 266.3 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();  
}
```

# 267 Longest Palindromic Substring

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

## 267.1 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()<=1)
        return s;

    int len = s.length();
    int maxLen = 1;
    boolean [][] dp = new boolean[len][len];

    String longest = null;
    for(int l=0; l<s.length(); l++){
        for(int i=0; i<len-l; i++){
            int j = i+l;
            if(s.charAt(i)==s.charAt(j) && (j-i<=2||dp[i+1][j-1])){
                dp[i][j]=true;

                if(j-i+1>maxLen){
                    maxLen = j-i+1;
                    longest = s.substring(i, j+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].

## 267.2 A Simple Algorithm

We can scan to both sides for each character. 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);
}

```

---

## 267.3 Manacher's Algorithm

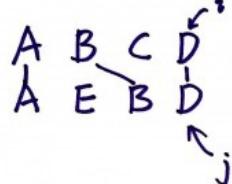
Manacher's algorithm is much more complicated to figure out, even though it will bring benefit of time complexity of  $O(n)$ . Since it is not typical, there is no need to waste time on that.

# 268 Longest Common Subsequence

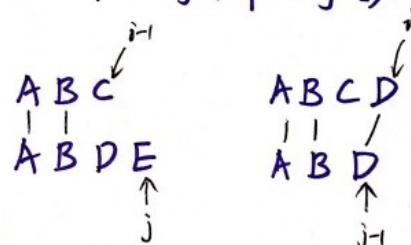
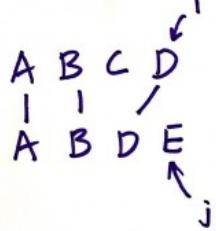
The longest common subsequence (LCS) problem is the problem of finding the longest subsequence common to all sequences in a set of sequences (often just two sequences).

## 268.1 Analysis

if  $a[i] == b[j]$ , then  $dp[i+1][j+1] = dp[i][j] + 1$



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



## 268.2 Java Solution

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

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

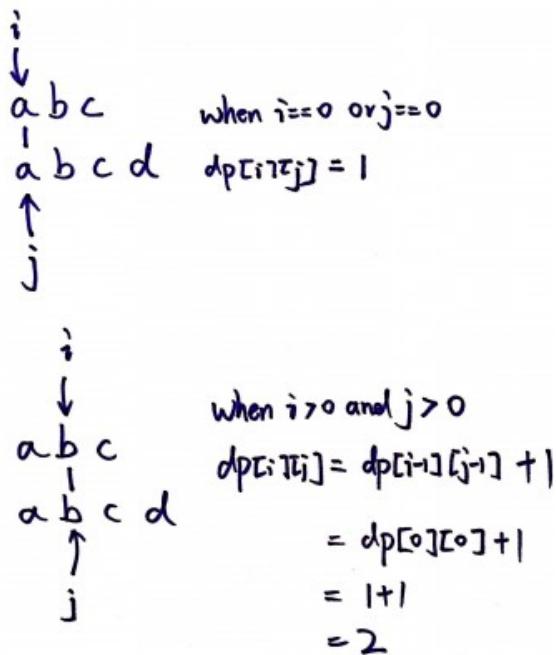
---

# 269 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.

## 269.1 Analysis

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



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

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

## 269.2 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  $dp[i][j-1]$ .

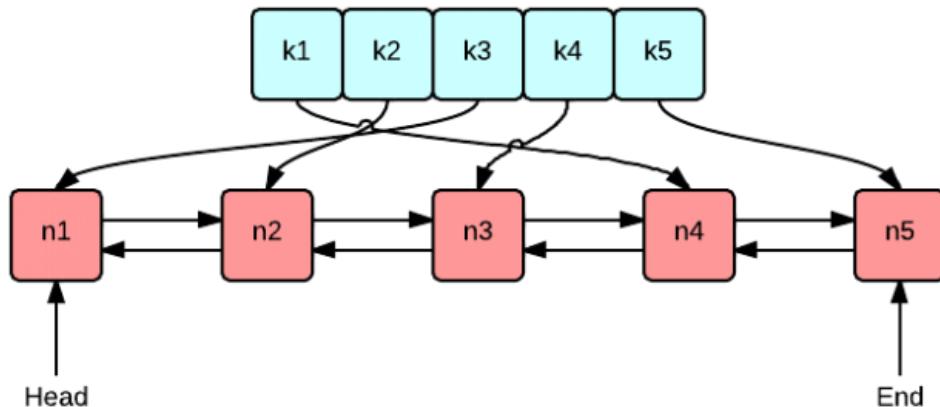
# 270 LRU Cache

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.

## 270.1 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).

## 270.2 Java Solution

Define a double linked list.

```
class Node{
    int key;
    int value;
    Node prev;
    Node next;

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

By analyzing the get and put, we can summarize there are 3 basic operations: 1) remove(Node t), 2) set-Head(Node t), and 3) HashMap put()/remove(). We only need to define the first two operations.

---

```

class LRUCache {
    HashMap<Integer, Node> map = null;
    int cap;
    Node head = null;
    Node tail = null;

    public LRUCache(int capacity) {
        this.map = new HashMap<>();
        this.cap = capacity;
    }

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

        Node t = map.get(key);

        remove(t);
        setHead(t);

        return t.value;
    }

    public void put(int key, int value) {
        if(map.containsKey(key)){
            Node t = map.get(key);
            t.value = value;

            remove(t);
            setHead(t);
        }else{
            if(map.size()>=cap){
                map.remove(tail.key);
                remove(tail);
            }

            Node t = new Node(key, value);
            setHead(t);
            map.put(key, t);
        }
    }

    //remove a node
    private void remove(Node t){
        if(t.prev!=null){
            t.prev.next = t.next;
        }else{
            head = t.next;
        }

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

```

```
//set a node to be head
private void setHead(Node t){
    if(head!=null){
        head.prev = t;
    }

    t.next = head;
    t.prev = null;
    head = t;

    if(tail==null){
        tail = head;
    }
}
```

---

# 271 Insert Delete GetRandom O(1)

Design a data structure that supports all following operations in O(1) time.

insert(val): Inserts an item val to the set if not already present. remove(val): Removes an item val from the set if present. getRandom: Returns a random element from current set of elements. Each element must have the same probability of being returned.

## 271.1 Java Solution

We can use two hashmaps to solve this problem. One uses value as keys and the other uses index as the keys.

```
class RandomizedSet {
    HashMap<Integer, Integer> valueMap;
    HashMap<Integer, Integer> idxMap;

    /** Initialize your data structure here. */
    public RandomizedSet() {
        valueMap = new HashMap<>();
        idxMap = new HashMap<>();
    }

    /** Inserts a value to the set. Returns true if the set did not already contain the specified
     * element. */
    public boolean insert(int val) {
        if(valueMap.containsKey(val)){
            return false;
        }

        valueMap.put(val, valueMap.size());
        idxMap.put(idxMap.size(), val);

        return true;
    }

    /** Removes a value from the set. Returns true if the set contained the specified element. */
    public boolean remove(int val) {
        if(valueMap.containsKey(val)){
            int idx = valueMap.get(val);
            valueMap.remove(val);
            idxMap.remove(idx);

            Integer tailElem = idxMap.get(idxMap.size());
            if(tailElem!=null){
                idxMap.put(idx,tailElem);
                valueMap.put(tailElem, idx);
            }

            return true;
        }

        return false;
    }
}
```

```
}

/** Get a random element from the set. */
public int getRandom() {
    if(valueMap.size()==0){
        return -1;
    }

    if(valueMap.size()==1){
        return idxMap.get(0);
    }

    Random r = new Random();
    int idx = r.nextInt(valueMap.size());

    return idxMap.get(idx);
}
```

---

# 272 Insert Delete GetRandom O(1) Duplicates allowed

Design a data structure that supports all following operations in average O(1) time.

Note: Duplicate elements are allowed.

---

```
insert(val): Inserts an item val to the collection.  
remove(val): Removes an item val from the collection if present.  
getRandom(): Returns a random element from current collection of elements.  
The probability of each element being returned is linearly related to the number of same value the  
collection contains.
```

---

## 272.1 Java Solution

This problem is similar to Insert Delete GetRandom O(1). We can use two maps. One tracks the index of the element, so that we can quickly insert and remove. The other maps tracks the order of each inserted element, so that we can randomly access any element in time O(1).

---

```
public class RandomizedCollection {  
    HashMap<Integer, HashSet<Integer>> map1;  
    HashMap<Integer, Integer> map2;  
    Random r;  
  
    /** Initialize your data structure here. */  
    public RandomizedCollection() {  
        map1 = new HashMap<Integer, HashSet<Integer>>();  
        map2 = new HashMap<Integer, Integer>();  
        r = new Random();  
    }  
  
    /** Inserts a value to the collection. Returns true if the collection did not already contain the  
     * specified element. */  
    public boolean insert(int val) {  
        //add to map2  
        int size2 = map2.size();  
        map2.put(size2+1, val);  
  
        if(map1.containsKey(val)){  
            map1.get(val).add(size2+1);  
            return false;  
        }else{  
            HashSet<Integer> set = new HashSet<Integer>();  
            set.add(size2+1);  
            map1.put(val, set);  
            return true;  
        }  
    }  
}
```

---

```

/** Removes a value from the collection. Returns true if the collection contained the specified
element. */
public boolean remove(int val) {
    if(map1.containsKey(val)){
        HashSet<Integer> set = map1.get(val);
        int toRemove = set.iterator().next();

        //remove from set of map1
        set.remove(toRemove);

        if(set.size()==0){
            map1.remove(val);
        }

        if(toRemove == map2.size()){
            map2.remove(toRemove);
            return true;
        }

        int size2 = map2.size();
        int key = map2.get(size2);

        HashSet<Integer> setChange = map1.get(key);
        setChange.remove(size2);
        setChange.add(toRemove);

        map2.remove(size2);
        map2.remove(toRemove);

        map2.put(toRemove, key);

        return true;
    }

    return false;
}

/** Get a random element from the collection. */
public int getRandom() {
    if(map1.size()==0)
        return -1;

    if(map2.size()==1){
        return map2.get(1);
    }

    return map2.get(r.nextInt(map2.size())+1); // nextInt() returns a random number in [0, n].
}

```

# 273 Design a Data Structure with Insert, Delete and GetMostFrequent of O(1)

Design a data structure that allows  $O(1)$  time complexity to insert, delete and get most frequent element.

## 273.1 Analysis

At first, a hash map seems to be good for insertion and deletion. But how to make getMostFrequent  $O(1)$ ? Regular sorting algorithm takes  $n \log n$ , so we can not use that. As a result we can use a linked list to track the maximum frequency.

## 273.2 Java Solution

---

```
import java.util.*;

class Node {
    int value;
    Node prev;
    Node next;
    HashSet<Integer> set;

    public Node(int v){
        value = v;
        set = new HashSet<Integer>();
    }

    public String toString(){
        return value + ":" + set.toString();
    }
}

public class FrequentCollection {

    HashMap<Integer, Node> map;
    Node head, tail;

    /** Initialize your data structure here. */
    public FrequentCollection() {
        map = new HashMap<Integer, Node>();
    }

    /**
     * Inserts a value to the collection.
     */
    public void insert(int val) {
        if(map.containsKey(val)){
            Node n = map.get(val);
            n.set.remove(val);
        }
    }

    /**
     * Returns the most frequent element.
     */
    public int getMostFrequent() {
        if(head == null)
            return -1;
        return head.value;
    }

    /**
     * Deletes the most frequent element.
     */
    public void deleteMostFrequent() {
        if(head == null)
            return;
        if(head.set.size() == 1)
            map.remove(head.value);
        else
            head.set.remove(head.value);
        head = head.next;
        if(head != null)
            head.prev = null;
    }
}
```

```

if(n.next!=null){
    n.next.set.add(val); // next + 1
    map.put(val, n.next);
}else{
    Node t = new Node(n.value+1);
    t.set.add(val);
    n.next = t;
    t.prev = n;
    map.put(val, t);
}

//update head
if(head.next!=null)
    head = head.next;
}else{
    if(tail==null||head==null){
        Node n = new Node(1);
        n.set.add(val);
        map.put(val, n);

        head = n;
        tail = n;
        return;
    }

    if(tail.value>1){
        Node n = new Node(1);
        n.set.add(val);
        map.put(val, n);
        tail.prev = n;
        n.next = tail;
        tail = n;
    }else{
        tail.set.add(val);
        map.put(val, tail);
    }
}

}

/** 
 * Removes a value from the collection.
 */
public void remove(int val) {
    Node n = map.get(val);
    n.set.remove(val);

    if(n.value==1){
        map.remove(val);
    }else{
        n.prev.set.add(val);
        map.put(val, n.prev);
    }
}

```

```
while(head!=null && head.set.size()==0){  
    head = head.prev;  
}  
  
}  
  
/** Get the most frequent element from the collection. */  
public int getMostFrequent() {  
    if(head==null)  
        return -1;  
    else  
        return head.set.iterator().next();  
}  
  
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    FrequentCollection fc = new FrequentCollection();  
    fc.insert(1);  
    fc.insert(2);  
    fc.insert(3);  
    fc.insert(2);  
    fc.insert(3);  
    fc.insert(3);  
    fc.insert(2);  
    fc.insert(2);  
  
    System.out.println(fc.getMostFrequent());  
    fc.remove(2);  
    fc.remove(2);  
    System.out.println(fc.getMostFrequent());  
  
}  
}
```

In the implementation above, we only add nodes to the list. We can also delete nodes that does not hold any elements.

## 274 Design Phone Directory

Design a Phone Directory which supports the following operations:

get: Provide a number which is not assigned to anyone. check: Check if a number is available or not. release: Recycle or release a number.

### 274.1 Java Solution 1

```
public class PhoneDirectory {
    int max;
    HashSet<Integer> set;
    LinkedList<Integer> queue;

    /** Initialize your data structure here
     * @param maxNumbers - The maximum numbers that can be stored in the phone directory. */
    public PhoneDirectory(int maxNumbers) {
        set = new HashSet<Integer>();
        queue = new LinkedList<Integer>();
        for(int i=0; i<maxNumbers; i++){
            queue.offer(i);
        }
        max=maxNumbers-1;
    }

    /** Provide a number which is not assigned to anyone.
     * @return - Return an available number. Return -1 if none is available. */
    public int get() {
        if(queue.isEmpty())
            return -1;

        int e = queue.poll();
        set.add(e);
        return e;
    }

    /** Check if a number is available or not. */
    public boolean check(int number) {
        return !set.contains(number) && number<=max;
    }

    /** Recycle or release a number. */
    public void release(int number) {
        if(set.contains(number)){
            set.remove(number);
            queue.offer(number);
        }
    }
}
```

## 275 Design Twitter

Design a simplified version of Twitter where users can post tweets, follow/unfollow another user and is able to see the 10 most recent tweets in the user's news feed. Your design should support the following methods:

postTweet(userId, tweetId): Compose a new tweet. getNewsFeed(userId): Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user herself. Tweets must be ordered from most recent to least recent. follow(followerId, followeeId): Follower follows a followee. unfollow(followerId, followeeId): Follower unfollows a followee.

### 275.1 Java Solution

---

```
class Wrapper{
    ArrayList<Integer> list;
    int index;

    public Wrapper(ArrayList<Integer> list, int index){
        this.list=list;
        this.index=index;
    }
}

public class Twitter {
    HashMap<Integer, HashSet<Integer>> userMap;//user and followees
    HashMap<Integer, ArrayList<Integer>> tweetMap;//user and tweets
    HashMap<Integer, Integer> orderMap; //tweet and order
    int order; //global order counter

    /** Initialize your data structure here. */
    public Twitter() {
        userMap = new HashMap<Integer, HashSet<Integer>>();
        tweetMap = new HashMap<Integer, ArrayList<Integer>>();
        orderMap = new HashMap<Integer, Integer>();
    }

    /** Compose a new tweet. */
    public void postTweet(int userId, int tweetId) {
        ArrayList<Integer> list = tweetMap.get(userId);
        if(list==null){
            list = new ArrayList<Integer>();
            tweetMap.put(userId, list);
        }
        list.add(tweetId);
        orderMap.put(tweetId, order++);
        follow(userId, userId);//follow himself
    }

    /** Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be
     * posted by users who the user followed or by the user herself. Tweets must be ordered from most
     * recent to least recent. */
    public List<Integer> getNewsFeed(int userId) {
```

```

    HashSet<Integer> set = userMap.get(userId);
    if(set==null)
        return new ArrayList<Integer>();

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

    //get all users' tweets
    for(int uid: set){
        if(tweetMap.get(uid)!=null && tweetMap.get(uid).size()>0)
            lists.add(tweetMap.get(uid));
    }

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

    PriorityQueue<Wrapper> queue = new PriorityQueue<Wrapper>(new Comparator<Wrapper>(){
        public int compare(Wrapper a, Wrapper b){
            return orderMap.get(b.list.get(b.index))-orderMap.get(a.list.get(a.index));
        }
    });

    for(ArrayList<Integer> list: lists){
        queue.offer(new Wrapper(list, list.size()-1));
    }

    while(!queue.isEmpty() && result.size()<10){
        Wrapper top = queue.poll();
        result.add(top.list.get(top.index));

        top.index--;
        if(top.index>=0)
            queue.offer(top);
    }

    return result;
}

/** Follower follows a followee. If the operation is invalid, it should be a no-op. */
public void follow(int followerId, int followeeId) {
    HashSet<Integer> set = userMap.get(followerId);
    if(set==null){
        set = new HashSet<Integer>();
        userMap.put(followerId, set);
    }
    set.add(followeeId);
}

/** Follower unfollows a followee. If the operation is invalid, it should be a no-op. */
public void unfollow(int followerId, int followeeId) {
    if(followerId==followeeId)
        return ;

    HashSet<Integer> set = userMap.get(followerId);
    if(set==null){
        return;
    }

    set.remove(followeeId);
}

```

}  
}

---

# 276 Single Number

The problem:

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

## 276.1 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;
}
```

---

## 276.2 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?

# 277 Single Number II

## 277.1 Problem

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

## 277.2 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;
}
```

---

# 278 Twitter Codility Problem Max Binary Gap

Problem: Get maximum binary Gap.

For example, 9's binary form is 1001, the gap is 2.

## 278.1 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  $O(n)$ .

## 278.2 Java Solution 2

---

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

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

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

        N = N & (N - 1);

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

    return len;
}
```

---

```
    }
    pre = curr;
}

return len;
}
```

---

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

# 279 Number of 1 Bits

## 279.1 Problem

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

For example, the 32-bit integer '11' has binary representation 00000000000000000000000000001011, so the function should return 3.

## 279.2 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;  
}
```

---

# 280 Reverse Bits

## 280.1 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](#)

## 280.2 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) {
        return n ^= (1 << i) | (1 << j);
    }

    return n;
}
```

---

# 281 Repeated DNA Sequences

## 281.1 Problem

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

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

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

## 281.2 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<>();
    if(s==null||s.length()<10){
        return result;
    }

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

    int hash=0;
    int mask = (1<<20) -1;

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

    for(int i=0; i<s.length(); i++){
        hash = (hash<<2) + dict.get(s.charAt(i));

        if(i>=9){
            hash&=mask;
            if(temp.contains(hash) && !added.contains(hash)){
                result.add(s.substring(i-9, i+1));
                added.add(hash);
            }
            temp.add(hash);
        }
    }

    return result;
}
```

}

## 282 Bitwise AND of Numbers Range

**282.1 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;
}
```

---

## 283 Sum of Two Integers

Calculate the sum of two integers a and b, but you are not allowed to use the operator + and -.

Example: Given a = 1 and b = 2, return 3.

### 283.1 Java Solution

Given two numbers a and b, a&b returns the number formed by '1' bits on a and b. When it is left shifted by 1 bit, it is the carry.

For example, given a=101 and b=111 (in binary), the a&b=101. a&b «1 = 1010.

ab is the number formed by different bits of a and b. a&b=10.

---

```
public int getSum(int a, int b) {  
  
    while(b!=0){  
        int c = a&b;  
        a=a^b;  
        b=c<<1;  
    }  
  
    return a;  
}
```

---

# 284 Counting Bits

Given a non negative integer number num. For every numbers i in the range  $0 \leq i \leq \text{num}$  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].

## 284.1 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;
}
```

---

## 284.2 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
:	:

For example

```

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;
}

```

## 285 Maximum Product of Word Lengths

Given a string array words, find the maximum value of  $\text{length}(\text{word}[i]) * \text{length}(\text{word}[j])$  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.

### 285.1 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;
}
```

---

## 286 Gray Code

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer  $n$  representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given  $n = 2$ , return [0,1,3,2]. Its gray code sequence is:

---

```
00 - 0
01 - 1
11 - 3
10 - 2
```

---

### 286.1 Java Solution

---

```
public List<Integer> grayCode(int n) {
    if(n==0){
        List<Integer> result = new ArrayList<Integer>();
        result.add(0);
        return result;
    }

    List<Integer> result = grayCode(n-1);
    int numToAdd = 1<<(n-1);

    for(int i=result.size()-1; i>=0; i--){ //iterate from last to first
        result.add(numToAdd+result.get(i));
    }

    return result;
}
```

---

## 287 UTF8 Validation

A character in UTF8 can be from 1 to 4 bytes long, subjected to the following rules:

For 1-byte character, the first bit is a 0, followed by its unicode code. For n-bytes character, the first n-bits are all one's, the n+1 bit is 0, followed by n-1 bytes with most significant 2 bits being 10. This is how the UTF-8 encoding would work:

Char. number range (hexadecimal)	UTF-8 octet sequence (binary)
0000 0000-0000 007F	0xxxxxxx
0000 0080-0000 07FF	110xxxxx 10xxxxxx
0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Given an array of integers representing the data, return whether it is a valid utf-8 encoding.

### 287.1 Java Solution

```
public boolean validUtf8(int[] data) {
    int i=0;
    int count=0;
    while(i<data.length){
        int v = data[i];
        if(count==0){
            if((v&240)==240 && (v&248)==240){
                count=3;
            }else if(((v&224)==224) && (v&240)==224){
                count=2;
            }else if((v&192)==192 && (v&224)==192){
                count=1;
            }else if((v|127)==127){
                count=0;
            }else{
                return false;
            }
        }else{
            if((v&128)==128 && (v&192)==128){
                count--;
            }else{
                return false;
            }
        }
        i++;
    }
    return count==0;
}
```

## 288 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.

### 288.1 Java Solution

---

```
public double myPow(double x, int n){  
    if(n==0)  
        return 1;  
  
    if(n<0){  
        return 1/helper(x, -n);  
    }  
  
    double v = helper(x, n/2);  
  
    if(n%2==0){  
        return v*v;  
    }else{  
        return v*v*x;  
    }  
}
```

---