

</>.....</>

# **JAVA Coding Interview Questions With Solution**

</>.....</>

# Contents

<b>1</b>	<b>Rotate Array in Java</b>	<b>13</b>
<b>2</b>	<b>Evaluate Reverse Polish Notation</b>	<b>17</b>
<b>3</b>	<b>Isomorphic Strings</b>	<b>21</b>
<b>4</b>	<b>Word Ladder</b>	<b>23</b>
<b>5</b>	<b>Word Ladder II</b>	<b>25</b>
<b>6</b>	<b>Median of Two Sorted Arrays</b>	<b>29</b>
<b>7</b>	<b>Kth Largest Element in an Array</b>	<b>31</b>
<b>8</b>	<b>Wildcard Matching</b>	<b>33</b>
<b>9</b>	<b>Regular Expression Matching in Java</b>	<b>35</b>
<b>10</b>	<b>Merge Intervals</b>	<b>39</b>
<b>11</b>	<b>Insert Interval</b>	<b>41</b>
<b>12</b>	<b>Two Sum</b>	<b>43</b>
<b>13</b>	<b>Two Sum II Input array is sorted</b>	<b>45</b>
<b>14</b>	<b>Two Sum III Data structure design</b>	<b>47</b>
<b>15</b>	<b>3Sum</b>	<b>49</b>
<b>16</b>	<b>4Sum</b>	<b>53</b>
<b>17</b>	<b>3Sum Closest</b>	<b>55</b>
<b>18</b>	<b>String to Integer (atoi)</b>	<b>57</b>
<b>19</b>	<b>Merge Sorted Array</b>	<b>59</b>

---

<b>21</b>	<b>Longest Valid Parentheses</b>	<b>63</b>
<b>22</b>	<b>Implement strStr()</b>	<b>65</b>
<b>23</b>	<b>Minimum Size Subarray Sum</b>	<b>69</b>
<b>24</b>	<b>Search Insert Position</b>	<b>73</b>
<b>25</b>	<b>Longest Consecutive Sequence</b>	<b>75</b>
<b>26</b>	<b>Valid Palindrome</b>	<b>77</b>
<b>27</b>	<b>ZigZag Conversion</b>	<b>81</b>
<b>28</b>	<b>Add Binary</b>	<b>83</b>
<b>29</b>	<b>Length of Last Word</b>	<b>85</b>
<b>30</b>	<b>Triangle</b>	<b>87</b>
<b>31</b>	<b>Contains Duplicate</b>	<b>89</b>
<b>32</b>	<b>Contains Duplicate II</b>	<b>91</b>
<b>33</b>	<b>Contains Duplicate III</b>	<b>93</b>
<b>34</b>	<b>Remove Duplicates from Sorted Array</b>	<b>95</b>
<b>35</b>	<b>Remove Duplicates from Sorted Array II</b>	<b>99</b>
<b>36</b>	<b>Longest Substring Without Repeating Characters</b>	<b>103</b>
<b>37</b>	<b>Longest Substring Which Contains 2 Unique Characters</b>	<b>105</b>
<b>38</b>	<b>Substring with Concatenation of All Words</b>	<b>109</b>
<b>39</b>	<b>Minimum Window Substring</b>	<b>111</b>
<b>40</b>	<b>Reverse Words in a String</b>	<b>113</b>
<b>41</b>	<b>Find Minimum in Rotated Sorted Array</b>	<b>115</b>
<b>42</b>	<b>Find Minimum in Rotated Sorted Array II</b>	<b>117</b>
<b>43</b>	<b>Search in Rotated Sorted Array</b>	<b>119</b>
<b>44</b>	<b>Search in Rotated Sorted Array II</b>	<b>121</b>

<b>46</b>	<b>Majority Element</b>	<b>125</b>
<b>47</b>	<b>Majority Element II</b>	<b>127</b>
<b>48</b>	<b>Remove Element</b>	<b>129</b>
<b>49</b>	<b>Largest Rectangle in Histogram</b>	<b>131</b>
<b>50</b>	<b>Longest Common Prefix</b>	<b>133</b>
<b>51</b>	<b>Largest Number</b>	<b>135</b>
<b>52</b>	<b>Simplify Path</b>	<b>137</b>
<b>53</b>	<b>Compare Version Numbers</b>	<b>139</b>
<b>54</b>	<b>Gas Station</b>	<b>141</b>
<b>55</b>	<b>Pascal's Triangle</b>	<b>143</b>
<b>56</b>	<b>Pascal's Triangle II</b>	<b>145</b>
<b>57</b>	<b>Container With Most Water</b>	<b>147</b>
<b>58</b>	<b>Candy</b>	<b>149</b>
<b>59</b>	<b>Trapping Rain Water</b>	<b>151</b>
<b>60</b>	<b>Count and Say</b>	<b>153</b>
<b>61</b>	<b>Search for a Range</b>	<b>155</b>
<b>62</b>	<b>Basic Calculator</b>	<b>157</b>
<b>63</b>	<b>Group Anagrams</b>	<b>159</b>
<b>64</b>	<b>Shortest Palindrome</b>	<b>161</b>
<b>65</b>	<b>Rectangle Area</b>	<b>163</b>
<b>66</b>	<b>Summary Ranges</b>	<b>165</b>
<b>67</b>	<b>Increasing Triplet Subsequence</b>	<b>167</b>
<b>68</b>	<b>Get Target Number Using Number List and Arithmetic Operations</b>	<b>169</b>
<b>69</b>	<b>Reverse Vowels of a String</b>	<b>171</b>

<b>71</b>	<b>Flip Game II</b>	<b>175</b>
<b>72</b>	<b>Move Zeroes</b>	<b>177</b>
<b>73</b>	<b>Valid Anagram</b>	<b>179</b>
<b>74</b>	<b>Group Shifted Strings</b>	<b>181</b>
<b>75</b>	<b>Top K Frequent Elements</b>	<b>183</b>
<b>76</b>	<b>Find Peak Element</b>	<b>185</b>
<b>77</b>	<b>Word Pattern</b>	<b>187</b>
<b>78</b>	<b>Set Matrix Zeroes</b>	<b>189</b>
<b>79</b>	<b>Spiral Matrix</b>	<b>193</b>
<b>80</b>	<b>Spiral Matrix II</b>	<b>197</b>
<b>81</b>	<b>Search a 2D Matrix</b>	<b>199</b>
<b>82</b>	<b>Search a 2D Matrix II</b>	<b>201</b>
<b>83</b>	<b>Rotate Image</b>	<b>205</b>
<b>84</b>	<b>Valid Sudoku</b>	<b>207</b>
<b>85</b>	<b>Minimum Path Sum</b>	<b>209</b>
<b>86</b>	<b>Unique Paths</b>	<b>211</b>
<b>87</b>	<b>Unique Paths II</b>	<b>213</b>
<b>88</b>	<b>Number of Islands</b>	<b>215</b>
<b>89</b>	<b>Number of Islands II</b>	<b>217</b>
<b>90</b>	<b>Surrounded Regions</b>	<b>219</b>
<b>91</b>	<b>Maximal Rectangle</b>	<b>223</b>
<b>92</b>	<b>Maximal Square</b>	<b>225</b>
<b>93</b>	<b>Word Search</b>	<b>227</b>
<b>94</b>	<b>Word Search II</b>	<b>229</b>

<b>96 Range Sum Query 2D Immutable</b>	<b>235</b>
<b>97 Longest Increasing Path in a Matrix</b>	<b>237</b>
<b>98 Implement a Stack Using an Array in Java</b>	<b>239</b>
<b>99 Add Two Numbers</b>	<b>243</b>
<b>100 Reorder List</b>	<b>245</b>
<b>101 Linked List Cycle</b>	<b>251</b>
<b>102 Copy List with Random Pointer</b>	<b>253</b>
<b>103 Merge Two Sorted Lists</b>	<b>257</b>
<b>104 Odd Even Linked List</b>	<b>259</b>
<b>105 Remove Duplicates from Sorted List</b>	<b>261</b>
<b>106 Remove Duplicates from Sorted List II</b>	<b>263</b>
<b>107 Partition List</b>	<b>265</b>
<b>108 LRU Cache</b>	<b>267</b>
<b>109 Intersection of Two Linked Lists</b>	<b>271</b>
<b>110 Remove Linked List Elements</b>	<b>273</b>
<b>111 Swap Nodes in Pairs</b>	<b>275</b>
<b>112 Reverse Linked List</b>	<b>277</b>
<b>113 Reverse Linked List II</b>	<b>279</b>
<b>114 Remove Nth Node From End of List</b>	<b>281</b>
<b>115 Implement Stack using Queues</b>	<b>283</b>
<b>116 Implement Queue using Stacks</b>	<b>285</b>
<b>117 Palindrome Linked List</b>	<b>287</b>
<b>118 Implement a Queue using an Array in Java</b>	<b>289</b>
<b>119 Delete Node in a Linked List</b>	<b>291</b>

# 1 Rotate Array in Java

You may have been using Java for a while. Do you think a simple Java array question can be a challenge? Let's use the following problem to test.

Problem: Rotate an array of  $n$  elements to the right by  $k$  steps. For example, with  $n = 7$  and  $k = 3$ , the array  $[1,2,3,4,5,6,7]$  is rotated to  $[5,6,7,1,2,3,4]$ . How many different ways do you know to solve this problem?

## 1.1 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()`.

## 1.2 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) {
```

---

## 1 Rotate Array in Java

---

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

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

---

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

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

---

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

---

### 1.3 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--;
    }
}
```

---

## 2 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:

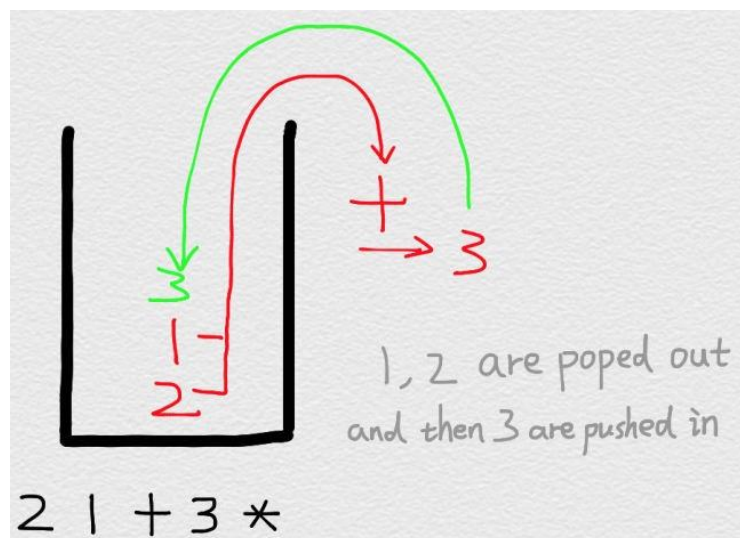
---

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

---

### 2.1 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 leet-code. Why?

---

```
public class Test {

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

## 2 Evaluate Reverse Polish Notation

---

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

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

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

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

    return returnValue;
}
```

---

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

### 2.2 Accepted Solution

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

---

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

        int returnValue = 0;
```

```
String operators = "+-*/";

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

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

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

return returnValue;
}
```

---

## 3 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.

### 3.1 Analysis

We need to define a method which accepts a map & a value, and returns the value's key in the map.

### 3.2 Java Solution

---

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

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

    if(s.length()==0 && t.length()==0)
        return true;

    HashMap<Character, Character> map = new HashMap<Character,Character>();
    for(int i=0; i<s.length(); i++){
        char c1 = s.charAt(i);
        char c2 = t.charAt(i);

        Character c = getKey(map, c2);
        if(c != null && c!=c1){
            return false;
        }else if(map.containsKey(c1)){
            if(c2 != map.get(c1))
                return false;
        }else{
            map.put(c1,c2);
        }
    }

    return true;
}
```

### 3 Isomorphic Strings

---

```
// a method for getting key of a target value
public Character getKey(HashMap<Character,Character> map, Character target){
    for (Map.Entry<Character,Character> entry : map.entrySet()) {
        if (entry.getValue().equals(target)) {
            return entry.getKey();
        }
    }

    return null;
}
```

---

## 4 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"]
```

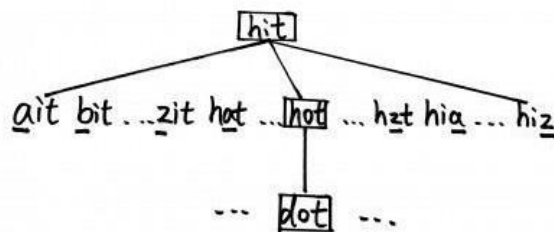
---

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

### 4.1 Analysis

UPDATED on 06/07/2015.

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



### 4.2 Java Solution

---

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

---

## 4 Word Ladder

---

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

        wordDict.add(endWord);

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

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

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

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

                        arr[i]=temp;
                    }
                }
            }

            return 0;
        }
    }
}
```

---



## 5 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"]
]
```

---

### 5.1 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.

### 5.2 Java Solution

---

```
class WordNode{
    String word;
    int numSteps;
    WordNode pre;

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

public class Solution {
    public List<List<String>> findLadders(String start, String end,
        Set<String> dict) {
        List<List<String>> result = new ArrayList<List<String>>();
```

## 5 Word Ladder II

---

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

dict.add(end);

int minStep = 0;

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

int preNumSteps = 0;

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

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

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

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

    preNumSteps = currNumSteps;

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

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

        arr[i]=temp;
    }

}

return result;
}
```

---

## 6 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  $O(\log(m+n))$ .*

### 6.1 Java Solution 1

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

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

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

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

---

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

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

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

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

    // Handle special cases
    if (aLen == 0)
        return B[bStart + k];
    if (bLen == 0)
        return A[aStart + k];
    if (k == 0)
        return A[aStart] < B[bStart] ? A[aStart] : B[bStart];

    int aMid = aLen * k / (aLen + bLen); // a's middle count
```

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

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

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

---

### 6.2 Java Solution 2

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

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

## 7 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,  $1 \leq k \leq \text{array's length}$ .

### 7.1 Java Solution 1 - Sorting

---

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

---

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

### 7.2 Java Solution 2 - Quick Sort

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

---

```
public int findKthLargest(int[] nums, int k) {  
    if (k < 1 || nums == null) {  
        return 0;  
    }  
  
    return getKth(nums.length - k + 1, nums, 0, nums.length - 1);  
}  
  
public int getKth(int k, int[] nums, int start, int end) {  
  
    int pivot = nums[end];  
  
    int left = start;  
    int right = end;  
  
    while (true) {  
  
        while (nums[left] < pivot && left < right) {  
            left++;  

```

## 7 Kth Largest Element in an Array

---

```
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  $O(n^2)$ .

### 7.3 Java Solution 3 - Heap

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

---

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

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

    return q.peek();
}
```

## 8 Wildcard Matching

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

### 8.1 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();
}
```

---



## 9 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",".a")` return true `isMatch("ab",".a")` return false `isMatch("aab","c*a*b")` return true

### 9.1 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 char of string, continue to match the remaining part.

### 9.2 Java Solution 1 (Short)

The following Java solution is accepted.

---

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

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

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

```
        int len = s.length();

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

---

### 9.3 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;
        }
    }
}
```

```
        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;
}
}
```

---

## 10 Merge Intervals

### Problem:

---

Given a collection of intervals, merge all overlapping intervals.

For example,

Given `[1,3],[2,6],[8,10],[15,18]`,

`return` `[1,6],[8,10],[15,18]`.

---

### 10.1 Thoughts of This Problem

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

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

### 10.2 Java Solution

---

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

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

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

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

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

## 10 Merge Intervals

---

```
Collections.sort(intervals, new IntervalComparator());

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

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

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

result.add(prev);

return result;
}

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

---

## 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:

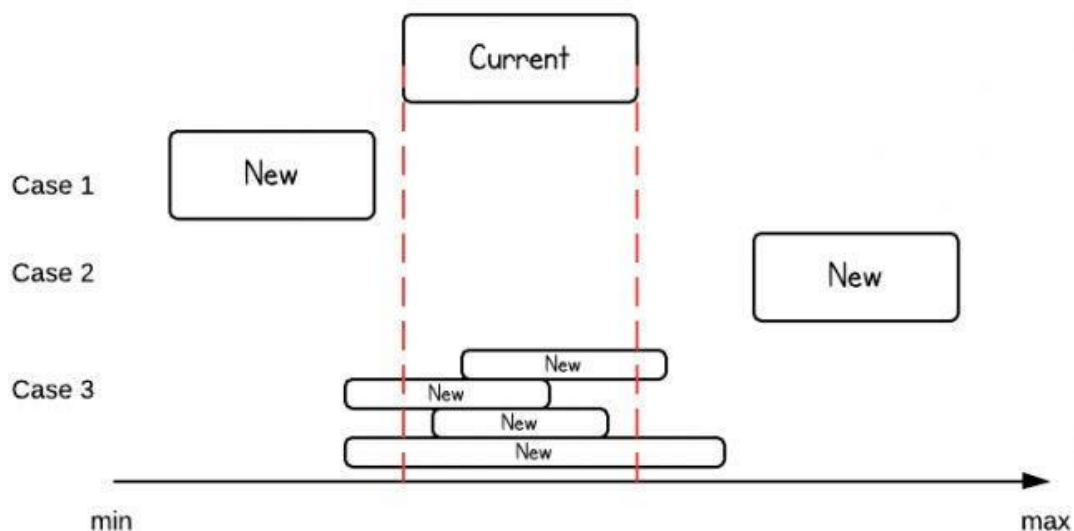
Given  $[1,2], [3,5], [6,7], [8,10], [12,16]$ , insert and merge  $[4,9]$  in as  $[1,2], [3,10], [12,16]$ .

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

---

### 11.1 Thoughts of This Problem

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



### 11.2 Java Solution

## 11 Insert Interval

---

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

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

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

        result.add(newInterval);

        return result;
    }
}
```

---

## 12 Two Sum

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

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

For example:

---

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

---

### 12.1 Naive Approach

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

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

---

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

---

Can we do better?

### 12.2 Better Solution

Use HashMap to store the target value.

---

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



## 12 Two Sum

---

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

---

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

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

## 13 Two Sum II Input array is sorted

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

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

---

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

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

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

    return null;
}
```

---

## 14 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
```

---

### 14.1 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;
            }
        }
    }
}
```

```
}  
}
```

---

## 15 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:

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

---

### 15.1 Naive Solution

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

---

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

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

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

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

                        each.add(num[i]);
                        each.add(num[j]);
                        each.add(num[k]);
                        result.add(each);
                        each.clear();
                    }
                }
            }
        }
        return result;
    }
}
```

```
        }
    }
}

    return result;
}
}
```

---

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

Result:

---

Submission Result: Output Limit Exceeded

---

## 15.2 Better Solution

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

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

---

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

    if (num.length < 3)
        return result;

    // sort array
    Arrays.sort(num);

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

            int negate = -num[i];

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

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

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

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

return result;
}
```

---

## 16 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  $\text{target} = 0$ .

A solution set is:

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

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

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

---

### 16.1 Thoughts

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

### 16.2 Java Solution

---

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

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

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

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

                if (sum > target) {
                    l--;
                } else if (sum < target) {
                    k++;
                } else if (sum == target) {
                    ArrayList<Integer> temp = new ArrayList<Integer>();
```



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

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

        k++;
        l--;
    }
}
}

return result;
}
```

---

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

---

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

---

## 17 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.  $(-1 + 2 + 1 = 2)$ .

---

### 17.1 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.

### 17.2 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--;
            }
        }
    }
}
```

## 17 3Sum Closest

---

```
    }  
  
    return result;  
}
```

---

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

## 18 String to Integer (atoi)

Implement atoi to convert a string to an integer.

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

### 18.1 Analysis

The following cases should be considered for this problem:

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

### 18.2 Java Solution

---

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

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

    char flag = '+';

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

    // use double to store result
    double result = 0;

    // calculate value
    while (str.length() > i && str.charAt(i) >= '0' && str.charAt(i) <= '9') {
```

## 18 String to Integer (atoi)

---

```
        i++;
    }

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

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

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

    return (int) result;
}
```

---

## 19 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.

### 19.1 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](#).

### 19.2 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--;
        }
    }
}
```

---

### 19.3 Java Solution 2

## 19 Merge Sorted Array

---

---

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

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

---

## 20 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.*

### 20.1 Analysis

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

### 20.2 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();
}
```

---



## 21 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.

### 21.1 Analysis

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

### 21.2 Java Solution

---

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

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

## 21 Longest Valid Parentheses

---

}

---

## 22 Implement strStr()

Problem:

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

### 22.1 Java Solution 1 - Naive

---

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

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

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

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

    return -1;
}
```

---

### 22.2 Java Solution 2 - KMP

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

---

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

## 22 Implement strStr()

---

```
        return 0;

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

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

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

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

    return -1;
}

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

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

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

```
    return next;  
}
```

---

## 23 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.

### 23.1 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.

### 23.2 Java Solution 1

A simple sliding window solution.

---

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

    int result = nums.length;

    int start=0;
    int sum=0;
    int i=0;
    boolean exists = false;

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

            result = Math.min(result, i-start);
            sum=sum-nums[start];
            start++;
        }else{
            if(i==nums.length)
```

## 23 Minimum Size Subarray Sum

---

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

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

---

### 23.3 Deprecated Java Solution

This solution works but it is less readable.

---

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

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

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

    for(int j=0; j<nums.length; ){
        if(i==j){
            if(nums[i]>=s){
                return 1; //if single elem is large enough
            }else{
                j++;

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

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

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

}

}

return result;
}
```

---



## 24 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
```

---

### 24.1 Solution 1

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

---

```
public class Solution {  
    public int searchInsert(int[] A, int target) {  
  
        if(A==null) return 0;  
  
        if(target <= A[0]) return 0;  
  
        for(int i=0; i<A.length-1; i++){  
            if(target > A[i] && target <= A[i+1]){  
                return i+1;  
            }  
        }  
  
        return A.length;  
    }  
}
```

---

### 24.2 Solution 2

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

---

```
public class Solution {
```

## 24 Search Insert Position

---

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

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

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

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

---

## 25 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.

### 25.1 Analysis

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

### 25.2 Java Solution

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

---

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

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

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

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

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

## 25 Longest Consecutive Sequence

---

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

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

    return max;
}
```

---

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

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

## 26 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.

### 26.1 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.

### 26.2 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)){
```

```
        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;
    }
}
}
```

---

## 26.3 Java Solution 2 - Using Stack

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

---

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

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

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

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

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

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

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

    return true;
}
```

---

## 26.4 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);
```

## 26 Valid Palindrome

---

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

    return true;
}

public static void main(String[] args) {
    String str = "A man, a plan, a canal: Panama";

    System.out.println(isValidPalindrome(str));
}
}
```

---



## 27 ZigZag Conversion

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

---

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

---

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

### 27.1 Java Solution

---

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

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

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

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

## 27 ZigZag Conversion

---

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

---

## 28 Add Binary

Given two binary strings, return their sum (also a binary string).  
For example, a = "11", b = "1", the return is "100".

### 28.1 Java Solution

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

---

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

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

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

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

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

## 28 Add Binary

---

```
        sb.append("1");  
    }  
  
    String reversed = sb.reverse().toString();  
    return reversed;  
}
```

---

## 29 Length of Last Word

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

### 29.1 Java Solution

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

---

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

---

## 30 Triangle

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

For example, given the following triangle

---

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

---

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

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

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

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

---

```
public class Solution {
    public int minimumTotal(ArrayList<ArrayList<Integer>> triangle) {

        int[] temp = new int[triangle.size()];
        int minTotal = Integer.MAX_VALUE;

        for(int i=0; i< temp.length; i++){
            temp[i] = Integer.MAX_VALUE;
        }

        if (triangle.size() == 1) {
            return Math.min(minTotal, triangle.get(0).get(0));
        }

        int first = triangle.get(0).get(0);

        for (int i = 0; i < triangle.size() - 1; i++) {
            for (int j = 0; j <= i; j++) {

                int a, b;
```

---

## 30 Triangle

---

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

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

    }

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

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

return minTotal;
}
}
```

---

## 30.2 Bottom-Up (Good Solution)

We can actually start from the bottom of the triangle.

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

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

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

    return total[0];
}
```

---

## 31 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.

### 31.1 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;  
}
```

---



## 32 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.

### 32.1 Java Solution 1

---

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

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

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

---

### 32.2 Java Solution 2 - Simplified

---

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

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

---

## 32 Contains Duplicate II

---

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

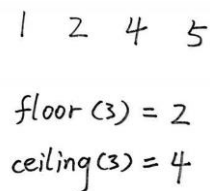
---

## 33 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  $k$ .

### 33.1 Java Solution 1

This solution uses a TreeSet.



1 2 4 5  
 $\text{floor}(3) = 2$   
 $\text{ceiling}(3) = 4$

The time complexity is  $O(n \log(k))$ .

---

```
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;  
}
```

---

## 33.2 Java Solution 2

Another solution that is easier to understand.

---

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

    SortedSet<Long> set = new TreeSet<Long>();

    for (int j = 0; j < nums.length; j++) {
        long leftBoundary = (long) nums[j] - t;
        long rightBoundary = (long) nums[j] + t + 1;
        SortedSet<Long> subSet = set.subSet(leftBoundary, rightBoundary);

        if (!subSet.isEmpty())
            return true;

        set.add((long) nums[j]);

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

    return false;
}
```

---

## 34 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].

### 34.1 Thoughts

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

### 34.2 Solution 1

---

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

    int j = 0;
    int i = 1;

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

    return j + 1;
}
```

---

This method returns the number of unique elements, but does not change the original array correctly. For example, if the input array is 1, 2, 2, 3, 3, the array will be

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

### 34.3 Solution 2

---

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

    int j = 0;
    int i = 1;

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

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

    return B;
}

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

---

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

### 34.4 Solution 3

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

---

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

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

---

## 35 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].

### 35.1 Naive Approach

Given the method signature "public int removeDuplicates(int[] A)", it seems that we should write a method that returns a integer and that's it. After typing the following solution:

---

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

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

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

            if(curr == pre){
                if(!flag){
                    flag = true;
                    continue;
                }else{
                    count++;
                }
            }else{
                pre = curr;
                flag = false;
            }
        }

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



## 35 Remove Duplicates from Sorted Array II

---

Online Judge returns:

---

Submission Result: Wrong Answer

Input: [1,1,1,2]

Output: [1,1,1]

Expected: [1,1,2]

---

So this problem also requires in-place array manipulation.

### 35.2 Correct Solution

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

---

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

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

        // index for updating
        int o = 1;

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

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

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

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

### 35.3 Better Solution

---

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

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

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

        return prev + 1;
    }
}
```

---

## 36 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 "bbbb" the longest substring is "b", with the length of 1.

### 36.1 Java Solution 1

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

---

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

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

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

## 36 Longest Substring Without Repeating Characters

---

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

return result;
}
```

---

### 36.2 Java Solution 2

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

The basic idea is using a hash table to track existing characters and their position. When a repeated character occurs, check from the previously repeated character. However, the time complexity is higher -  $O(n^3)$ .

---

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

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

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

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

---

Consider the following simple example.

---

abcda

---

When loop hits the second "a", the HashMap contains the following:

---

a	0
b	1
c	2
d	3

---

The index *i* is set to 0 and incremented by 1, so the loop start from second element again.

## 37 Longest Substring Which Contains 2 Unique Characters

This is a problem asked by Google.

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

### 37.1 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++;
            }
        }
    }
}
```

## 37 Longest Substring Which Contains 2 Unique Characters

---

```
    }  
}  
  
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?

### 37.2 Solution for K Unique Characters

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

---

```
public int lengthOfLongestSubstringKDistinct(String s, int k) {  
    int max=0;  
    HashMap<Character,Integer> map = new HashMap<Character, Integer>();  
    int start=0;  
  
    for(int i=0; i<s.length(); i++){  
        char c = s.charAt(i);  
        if(map.containsKey(c)){  
            map.put(c, map.get(c)+1);  
        }else{  
            map.put(c,1);  
        }  
  
        if(map.size()>k){  
            max = Math.max(max, i-start);  
  
            while(map.size()>k){  
                char t = s.charAt(start);  
                int count = map.get(t);  
                if(count>1){  
                    map.put(t, count-1);  
                }else{  
                    map.remove(t);  
                }  
                start++;  
            }  
        }  
  
        max = Math.max(max, s.length()-start);  
  
    return max;  
}
```

Time is  $O(n)$ .

## 38 Substring with Concatenation of All Words

You are given a string, *s*, and a list of words, *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 *words* exactly once and without any intervening characters.

For example, given: *s*="barfoothefoobarman" & *words*=["foo", "bar"], return [0,9].

### 38.1 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.

### 38.2 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 to total 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++;

        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;
}
```

---

## 39 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".

### 39.1 Java Solution

---

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

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

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

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

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

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

### 39 Minimum Window Substring

---

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

---

## 40 Reverse Words in a String

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

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

### 40.1 Java Solution

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

---

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

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

---

## 41 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.

### 41.1 Analysis

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

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

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

### 41.2 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]) {
```

## 41 Find Minimum in Rotated Sorted Array

---

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

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

---

### 41.3 Java Solution 2 - Iteration

---

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

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

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

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

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

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

    return nums[left];
}
```

---

## 42 Find Minimum in Rotated Sorted Array II

### 42.1 Problem

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

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

### 42.2 Java Solution

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

---

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

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

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

## 42 Find Minimum in Rotated Sorted Array II

---

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

---



## 43 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.

### 43.1 Java Solution 1- Recursive

---

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

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

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

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

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

---

### 43.2 Java Solution 2 - Iterative

### 43 Search in Rotated Sorted Array

---

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

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

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

return -1;
}
```

---

## 44 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.

### 44.1 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;
}
```

---

## 45 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.

### 45.1 Analysis

UPDATED ON 6/17/2015

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

### 45.2 Java Solution

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

---

```
class Node {
    int value;
    int min;
    Node next;

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

---

```
class MinStack {
    Node head;

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

---

```
        head = temp;
    }
}

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

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

    return head.value;
}

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

    return head.min;
}
}
```

---

## 46 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.)

### 46.1 Java Solution 1 - Naive

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

---

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

        Arrays.sort(num);

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

        return 0;
    }
}
```

---

### 46.2 Java Solution 2 - Much Simpler

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

## 46 Majority Element

---

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

    Arrays.sort(num);
    return num[num.length / 2];
}
```

---

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

---

```
public int majorityElement(int[] nums) {
    int result = 0, count = 0;

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

    return result;
}
```

---

## 47 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.

### 47.1 Java Solution 1 - Using a Counter

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

---

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

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

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

    return result;
}
```

---

### 47.2 Java Solution 2

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

Check out [Majority Element I](#).

---

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

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



## 47 Majority Element II

---

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

    c1 = c2 = 0;

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

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

    return result;
}
```

---

## 48 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.)

### 48.1 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++;
        }

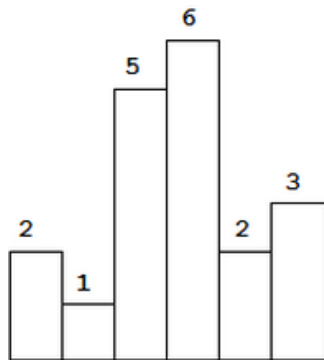
        j++;
    }

    return i;
}
```

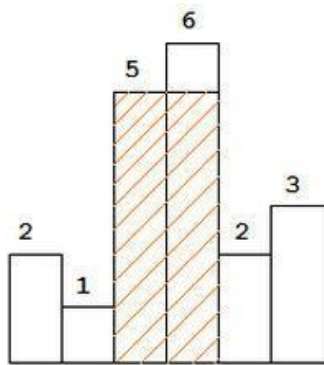
---

## 49 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.

### 49.1 Analysis

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

## 49.2 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;
}
```

---

## 50 Longest Common Prefix

### 50.1 Problem

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

### 50.2 Analysis

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

### 50.3 Java Solution

---

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

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

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

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

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

---

## 51 Largest Number

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

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

### 51.1 Analysis

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

### 51.2 Java Solution

---

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

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

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

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

    return sb.toString();
}
```

---

## 52 Simplify Path

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

For example,

---

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

---

### 52.1 Java Solution

---

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

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

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

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

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

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

    return back == 0 ? "/" : result.join("");
}
```

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

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

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

return sb.toString();
}
```

---



## 53 Compare Version Numbers

### 53.1 Problem

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

---

0.1 < 1.1 < 1.2 < 13.37

---

### 53.2 Java Solution

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

---

```
public int compareVersion(String version1, String version2) {
    String[] arr1 = version1.split("\\.");
    String[] arr2 = version2.split("\\.");

    int i=0;
    while(i<arr1.length || i<arr2.length){
        if(i<arr1.length && i<arr2.length){
            if(Integer.parseInt(arr1[i]) < Integer.parseInt(arr2[i])){
                return -1;
            }else if(Integer.parseInt(arr1[i]) > Integer.parseInt(arr2[i])){
                return 1;
            }
        }
        else if(i<arr1.length){
            if(Integer.parseInt(arr1[i]) != 0){
                return 1;
            }
        }
        else if(i<arr2.length){
            if(Integer.parseInt(arr2[i]) != 0){
                return -1;
            }
        }

        i++;
    }
}
```

## 53 Compare Version Numbers

---

}

---

## 54 Gas Station

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

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

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

### 54.1 Analysis

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

Proof of fact 2:

---

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

So to reach C from A,  $gas[A]$  must  $\geq cost[A]$ .

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

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

---

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

### 54.2 Java Solution

---

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

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

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

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

---

## 55 Pascal's Triangle

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

---

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

---

### 55.1 Java Solution

---

```
public ArrayList<ArrayList<Integer>> generate(int numRows) {
    ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
    if (numRows <= 0)
        return result;

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

    for (int i = 2; i <= numRows; i++) {
        ArrayList<Integer> cur = new ArrayList<Integer>();

        cur.add(1); //first
        for (int j = 0; j < pre.size() - 1; j++) {
            cur.add(pre.get(j) + pre.get(j + 1)); //middle
        }
        cur.add(1); //last

        result.add(cur);
        pre = cur;
    }

    return result;
}
```

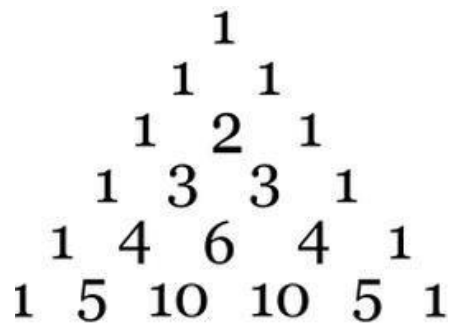
---

## 56 Pascal's Triangle II

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

### 56.1 Analysis

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



### 56.2 Java Solution

---

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

    if (rowIndex < 0)
        return result;

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

---

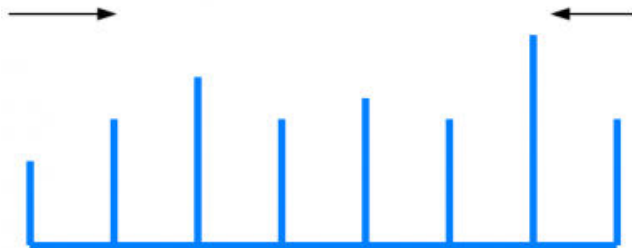
## 57 Container With Most Water

### 57.1 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.

### 57.2 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.



### 57.3 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],  

```

## 57 Container With Most Water

---

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

return max;
}
```

---



## 58 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?

### 58.1 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](#).

### 58.2 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;
}
```

---

## 59 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.

### 59.1 Analysis

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

### 59.2 Java Solution

---

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

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

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

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

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

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

    return result;
}
```

## 59 Trapping Rain Water

---

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

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

return result;
}
```

---

## 60 Count and Say

### 60.1 Problem

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

---

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

---

Given an integer  $n$ , generate the  $n$ th sequence.

### 60.2 Java Solution

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

---

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

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

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

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

---

## 60 Count and Say

---

```
    return result;  
}
```

---

## 61 Search for a Range

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

### 61.1 Analysis

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

### 61.2 Java Solution

---

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

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

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

    return arr;
}

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

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

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

## 61 Search for a Range

---

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

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

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

---



## 62 Basic Calculator

Implement a basic calculator to evaluate a simple expression string.

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

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

### 62.1 Analysis

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

### 62.2 Java Solution

---

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

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

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

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

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

            if (arr[i] != ')') {
```

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

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

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

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

return temp;
}
```

---

## 63 Group Anagrams

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

### 63.1 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.

### 63.2 Java Solution

---

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

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

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

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

    result.addAll(map.values());

    return result;
}
```

### 63.3 Time Complexity

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

## 64 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".

### 64.1 Analysis

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.

### 64.2 Java Solution

---

```
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;
```

## 64 Shortest Palindrome

---

```
//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();
}
```

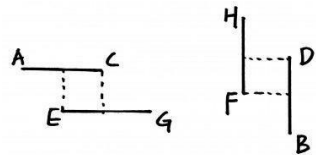
---

## 65 Rectangle Area

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

### 65.1 Analysis

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



### 65.2 Java Solution

---

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

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

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

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

---

## 66 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"].

### 66.1 Analysis

### 66.2 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];
        }
    }
}
```



## 66 Summary Ranges

---

```
        pre = nums[i];  
    }  
  
    return result;  
}
```

---

## 67 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.

### 67.1 Analysis

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

### 67.2 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;
}
```

---

## 68 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  $(1+2)*(3+4)=21$

### 68.1 Analysis

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

### 68.2 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) {
```

```
        return result;
    }

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

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

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

    return result;
}
```

---

## 69 Reverse Vowels of a String

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

### 69.1 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++;
    }
}
```

## 69 Reverse Vowels of a String

---

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

---

## 70 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.

### 70.1 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;
}
```

---

## 71 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 "+--".

### 71.1 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;
}
```



## 71.2 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.

## 72 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]`.

### 72.1 Java Solution

---

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

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

---

## 73 Valid Anagram

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

### 73.1 Java Solution 1

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

---

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

---

### 73.2 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) {
```

---

### 73 Valid Anagram

---

```
        return false;

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

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

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

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

    return true;
}
```

---

## 74 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"]
]
```

---

### 74.1 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);
            }
        }
    }

    return result;
}
```

## 74 Group Shifted Strings

---

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

---

## 75 Top K Frequent Elements

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

### 75.1 Java Solution

We can solve this problem by using a 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;  
    }  
}  
}
```

---



## 76 Find Peak Element

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

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

### 76.1 Thoughts

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

### 76.2 Java Solution

---

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

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

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

        return index;
    }
}
```

---

## 77 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.

### 77.1 Java Solution

---

```
public boolean wordPattern(String pattern, String str) {
    String[] arr = str.split(" ");

    if (pattern.length() != arr.length)
        return false;

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

    for (int i = 0; i < arr.length; i++) {
        char c = pattern.charAt(i);
        String s = arr[i];

        if (map.containsKey(c)) {
            if (!map.get(c).equals(s))
                return false;
        } else {
            if (map.containsValue(s))
                return false;

            map.put(c, s);
        }
    }

    return true;
}
```

---

## 78 Set Matrix Zeroes

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

### 78.1 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

## 78.2 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;
}
}
```

---

## 79 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].

### 79.1 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;
}
}
```

---

## 79.2 Java Solution 2

We can also recursively solve this problem. The solution's performance is not better than Solution 0 or as clear as Solution 1. Therefore, Solution 1 should be preferred.

---

```
public class Solution {
    public ArrayList<Integer> spiralOrder(int[][] matrix) {
        if(matrix==null || matrix.length==0)
            return new ArrayList<Integer>();

        return spiralOrder(matrix,0,0,matrix.length,matrix[0].length);
    }
}
```

---

```
public ArrayList<Integer> spiralOrder(int [][] matrix, int x, int y, int
    m, int n) {
    ArrayList<Integer> result = new ArrayList<Integer>();

    if(m<=0||n<=0)
        return result;

    //only one element left
    if(m==1&&n==1) {
        result.add(matrix[x][y]);
        return result;
    }

    //top - move right
    for(int i=0;i<n-1;i++){
        result.add(matrix[x][y++]);
    }

    //right - move down
    for(int i=0;i<m-1;i++){
        result.add(matrix[x++][y]);
    }

    //bottom - move left
    if(m>1){
        for(int i=0;i<n-1;i++){
            result.add(matrix[x][y--]);
        }
    }

    //left - move up
    if(n>1){
        for(int i=0;i<m-1;i++){
            result.add(matrix[x--][y]);
        }
    }

    if(m==1||n==1)
        result.addAll(spiralOrder(matrix, x, y, 1, 1));
    else
        result.addAll(spiralOrder(matrix, x+1, y+1, m-2, n-2));

    return result;
}
}
```

---



## 80 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]
]
```

---

### 80.1 Java Solution

---

```
public int[][] generateMatrix(int n) {
    int total = n*n;
    int[][] result= new int[n][n];

    int x=0;
    int y=0;
    int step = 0;

    for(int i=0;i<total;){
        while(y+step<n){
            i++;
            result[x][y]=i;
            y++;

        }
        y--;
        x++;

        while(x+step<n){
            i++;
            result[x][y]=i;
            x++;

        }
        x--;
        y--;

        while(y>=0+step){
            i++;
            result[x][y]=i;
            y--;

        }
        y++;
        x--;

        step++;
    }
    return result;
}
```

```
        result[x][y]=i;
        y--;
    }
    y++;
    x--;
    step++;

    while(x>=0+step){
        i++;
        result[x][y]=i;
        x--;
    }
    x++;
    y++;
}

return result;
}
```

---

## 81 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.

### 81.1 Java Solution

This is a typical problem of binary search.

You may try to solve this problem by finding the row first and then the column. There is no need to do that. Because of the matrix's special features, the matrix can be considered as a sorted array. Your goal is to find one element in this sorted array by using binary search.

---

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if(matrix==null || matrix.length==0 || matrix[0].length==0)
            return false;

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

        int start = 0;
        int end = m*n-1;

        while(start<=end){
            int mid=(start+end)/2;
            int midX=mid/n;
            int midY=mid%n;

            if(matrix[midX][midY]==target)
                return true;
        }
    }
}
```

---

## 81 Search a 2D Matrix

---

```
        if(matrix[midX][midY]<target){
            start=mid+1;
        }else{
            end=mid-1;
        }
    }

    return false;
}
}
```

---

## 82 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.

### 82.1 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]){
```

```
    }
}

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;
}
```

---

## 82.2 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 false;  
}
```

---

## 83 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?

### 83.1 Naive Solution

In the following solution, a new 2-dimension array is created to store the rotated matrix, and the result is assigned to the matrix at the end. This is **WRONG!** Why?

---

```
public class Solution {
    public void rotate(int[][] matrix) {
        if(matrix == null || matrix.length==0)
            return ;

        int m = matrix.length;

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

        for(int i=0; i<m; i++){
            for(int j=0; j<m; j++){
                result[j][m-1-i] = matrix[i][j];
            }
        }

        matrix = result;
    }
}
```

---

The problem is that Java is pass by value not by reference! "matrix" is just a reference to a 2-dimension array. If "matrix" is assigned to a new 2-dimension array in the method, the original array does not change. Therefore, there should be another loop to assign each element to the array referenced by "matrix". Check out "[Java pass by value](#)."

---

```
public class Solution {
    public void rotate(int[][] matrix) {
        if(matrix == null || matrix.length==0)
            return ;

        int m = matrix.length;
```



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

for(int i=0; i<m; i++){
    for(int j=0; j<m; j++){
        result[j][m-1-i] = matrix[i][j];
    }
}

for(int i=0; i<m; i++){
    for(int j=0; j<m; j++){
        matrix[i][j] = result[i][j];
    }
}
}
```

---

## 83.2 In-place Solution

By using the relation " $\text{matrix}[i][j] = \text{matrix}[n-1-j][i]$ ", 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;
        }
    }
}
```

---

## 84 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

### 84.1 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];
```

```
        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;
}
```

---

## 85 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.

### 85.1 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;
}
```

---

### 85.2 Java Solution 2: Dynamic Programming

---

```
public int minPathSum(int[][] grid) {
    if(grid == null || grid.length==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];
}
```

---

## 86 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?

### 86.1 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;  
}
```

---

### 86.2 Java Solution 2 - Dynamic Programming

---

```
public int uniquePaths(int m, int n) {  
    if(m==0 || n==0) return 0;  
    // ...  
}
```

---

```
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];
}
```

---

## 87 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 3x3 grid as illustrated below,

---

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

---

the total number of unique paths is 2.

### 87.1 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];
        }
    }
}
```



```
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];
}
```

---

## 88 Number of Islands

Given a 2-d grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

---

```
11110
11010
11000
00000
```

---

Answer: 1

Example 2:

---

```
11000
11000
00100
00011
```

---

Answer: 3

### 88.1 Java Solution

The basic idea of the following solution is merging adjacent lands, and the merging should be done recursively.

---

```
public int numIslands(char[][] grid) {
    if(grid==null||grid.length==0||grid[0].length==0)
        return 0;

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

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

## 88 Number of Islands

---

```
        return count;
    }

    public void merge(char[][] grid, int i, int j){
        if(i<0||j<0||i>=grid.length||j>=grid[0].length)
            return;

        if(grid[i][j]=='1'){
            grid[i][j]='0';

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

---

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

## 89 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.

### 89.1 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[arr[i]];
    }
    return i;
}
```

---

## 90 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
```

---

### 90.1 Analysis

This problem is similar to [Number of Islands](#). In this problem, only the cells on the borders can not be surrounded. So we can first merge those O's on the borders like in [Number of Islands](#) and replace O's with '#', and then scan the board and replace all O's left (if any).

### 90.2 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] == 'O'){
            merge(board, i, 0);
        }
    }
}
```

```
        merge(board, i, n-1);
    }
}

//merge O's on top & bottom boarder
for(int j=0; j<n; j++){
    if(board[0][j] == 'O'){
        merge(board, 0, j);
    }

    if(board[m-1][j] == 'O'){
        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] == 'O'){
            board[i][j] = 'X';
        } else if(board[i][j] == '#'){
            board[i][j] = 'O';
        }
    }
}
}

public void merge(char[][] board, int i, int j){
    if(i<0 || i>=board.length || j<0 || j>=board[0].length)
        return;

    if(board[i][j] != 'O')
        return;

    board[i][j] = '#';

    merge(board, i-1, j);
    merge(board, i+1, j);
    merge(board, i, j-1);
    merge(board, i, j+1);
}
```

---

This solution causes `java.lang.StackOverflowError`, because for a large board, too many method calls are pushed to the stack and causes the overflow.

### 90.3 Breath-first Search

Instead we use a queue to do breath-first search.

```
public class Solution {
    // use a queue to do BFS
    private Queue<Integer> queue = new LinkedList<Integer>();

    public void solve(char[][] board) {
        if (board == null || board.length == 0)
            return;

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

        // merge O's on left & right boarder
        for (int i = 0; i < m; i++) {
            if (board[i][0] == 'O') {
                bfs(board, i, 0);
            }

            if (board[i][n - 1] == 'O') {
                bfs(board, i, n - 1);
            }
        }

        // merge O's on top & bottom boarder
        for (int j = 0; j < n; j++) {
            if (board[0][j] == 'O') {
                bfs(board, 0, j);
            }

            if (board[m - 1][j] == 'O') {
                bfs(board, m - 1, j);
            }
        }

        // process the board
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i][j] == 'O') {
                    board[i][j] = 'X';
                } else if (board[i][j] == '#') {
                    board[i][j] = 'O';
                }
            }
        }
    }

    private void bfs(char[][] board, int i, int j) {
        int n = board[0].length;

        // fill current first and then its neighbors
    }
}
```



```
while (!queue.isEmpty()) {
    int cur = queue.poll();
    int x = cur / n;
    int y = cur % n;

    fillCell(board, x - 1, y);
    fillCell(board, x + 1, y);
    fillCell(board, x, y - 1);
    fillCell(board, x, y + 1);
}
}

private void fillCell(char[][] board, int i, int j) {
    int m = board.length;
    int n = board[0].length;
    if (i < 0 || i >= m || j < 0 || j >= n || board[i][j] != 'O')
        return;

    // add current cell to queue & then process its neighbors in bfs
    queue.offer(i * n + j);
    board[i][j] = '#';
}
}
```

---

## 91 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.

### 91.1 Analysis

This problem can be converted to the "[Largest Rectangle in Histogram](#)" problem.

### 91.2 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;
```

## 91 Maximal Rectangle

---

```
while (i < height.length) {
    if (stack.isEmpty() || height[stack.peek()] <= height[i]) {
        stack.push(i++);
    } else {
        int t = stack.pop();
        max = Math.max(max, height[t]
            * (stack.isEmpty() ? i : i - stack.peek() - 1));
    }
}

return max;
}
```

---

## 92 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.

### 92.1 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.

### 92.2 Java Solution

---

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

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

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

    //top row
    for (int i = 0; i < m; i++) {
        t[i][0] = Character.getNumericValue(matrix[i][0]);
    }

    //left column
    for (int j = 0; j < n; j++) {
        t[0][j] = Character.getNumericValue(matrix[0][j]);
    }

    //cells inside
```

```
for (int j = 1; j < n; j++) {
    if (matrix[i][j] == '1') {
        int min = Math.min(t[i - 1][j], t[i - 1][j - 1]);
        min = Math.min(min, t[i][j - 1]);
        t[i][j] = min + 1;
    } else {
        t[i][j] = 0;
    }
}

int max = 0;
//get maximal length
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (t[i][j] > max) {
            max = t[i][j];
        }
    }
}

return max * max;
}
```

---

## 93 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 = "ABCCED", ->returns true, word = "SEE", ->returns true, word = "ABCB", ->returns false.

### 93.1 Analysis

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

### 93.2 Java Solution

---

```
public boolean exist(char[][] board, String word) {
    int m = board.length;
    int n = board[0].length;

    boolean result = false;
    for(int i=0; i<m; i++){
        for(int j=0; j<n; j++){
            if(dfs(board,word,i,j,0)){
                result = true;
            }
        }
    }

    return result;
}

public boolean dfs(char[][] board, String word, int i, int j, int k){
```

```
int 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;
}
```

---

## 94 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"].

### 94.1 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;
                }
            }
        }
    }
}
```

---



```
        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;
}
```

---

## 94.2 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>();
    }
```

---

```

        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;
    }
}
```

---

## 95 Integer Break

Given a positive integer  $n$ , break it into the sum of at least two positive integers and maximize the product of those integers. Return the maximum product you can get.

For example, given  $n = 2$ , return 1 ( $2 = 1 + 1$ ); given  $n = 10$ , return 36 ( $10 = 3 + 3 + 4$ ).

### 95.1 Java Solution 1 - Dynamic Programming

Let  $dp[i]$  to be the max production value for breaking the number  $i$ . Since  $dp[i+j]$  can be  $i*j$ ,  $dp[i+j] = \max(\max(dp[i], i) * \max(dp[j], j)), dp[i+j])$ .

---

```
public int integerBreak(int n) {
    int[] dp = new int[n+1];

    for(int i=1; i<n; i++){
        for(int j=1; j<i+1; j++){
            if(i+j<=n) {
                dp[i+j]=Math.max(Math.max(dp[i],i)*Math.max(dp[j],j), dp[i+j]);
            }
        }
    }

    return dp[n];
}
```

---

### 95.2 Java Solution 2 - Using Regularities

If we see the breaking result for some numbers, we can see repeated pattern like the following:

---

```
2 -> 1*1
3 -> 1*2
4 -> 2*2
5 -> 3*2
6 -> 3*3
7 -> 3*4
8 -> 3*3*2
9 -> 3*3*3
10 -> 3*3*4
```

We only need to find how many 3's we can get when  $n > 4$ . If  $n$

---

```
public int integerBreak(int n) {  
  
    if (n==2) return 1;  
    if (n==3) return 2;  
    if (n==4) return 4;  
  
    int result=1;  
    if (n%3==0) {  
        int m = n/3;  
        result = (int) Math.pow(3, m);  
    } else if (n%3==2) {  
        int m=n/3;  
        result = (int) Math.pow(3, m) * 2;  
    } else if (n%3==1) {  
        int m=(n-4)/3;  
        result = (int) Math.pow(3, m) * 4;  
    }  
  
    return result;  
}
```

---

## 96 Range Sum Query 2D Immutable

Given a 2D matrix `matrix`, find the sum of the elements inside the rectangle defined by its upper left corner `(row1, col1)` and lower right corner `(row2, col2)`.

### 96.1 Analysis

Since the assumption is that there are many calls to `sumRegion` method, we should use some extra space to store the intermediate results. Here we define an array `sum[][]` which stores the sum value from `(0,0)` to the current cell.

### 96.2 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) {
```

```
        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;
}
}
```

---

## 97 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

### 97.1 Java Solution 1 - DFS

This solution is over time limit.

---

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

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

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

        return longest;
    }

    public void helper(int[][] matrix, int i, int j, int len){

        if(i-1>=0 && matrix[i-1][j]>matrix[i][j]){
            longest = Math.max(longest, len+1);
            helper(matrix, i-1, j, len+1);
        }

        if(i+1<matrix.length && matrix[i+1][j]>matrix[i][j]){
            longest = Math.max(longest, len+1);
            helper(matrix, i+1, j, len+1);
        }

        if(j-1>=0 && matrix[i][j-1]>matrix[i][j]){
            longest = Math.max(longest, len+1);
            helper(matrix, i, j-1, len+1);
        }
    }
}
```



```
        if(j+1<matrix[0].length && matrix[i][j+1]>matrix[i][j]){
            longest = Math.max(longest, len+1);
            helper(matrix, i, j+1, len+1);
        }
    }
}
```

---

## 97.2 Java Solution - Optimized

---

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

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

        int[][] mem = new int[matrix.length][matrix[0].length];
        int longest=0;

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

        return longest;
    }

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

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

            if(x>=0&&y>=0&&x<matrix.length&&y<matrix[0].length&&matrix[x][y]>matrix[i][j]){
                mem[i][j]=Math.max(mem[i][j], dfs(matrix, x, y, mem));
            }
        }

        return ++mem[i][j];
    }
}
```

---

## 98 Implement a Stack Using an Array in Java

This post shows how to implement a stack by using an array.

The requirements of the stack are: 1) the stack has a constructor which accept a number to initialize its size, 2) the stack can hold any type of elements, 3) the stack has a push() and a pop() method.

I remember there is a similar example in the "Effective Java" book written by Joshua Bloch, but not sure how the example is used. So I just write one and then read the book, and see if I miss anything.

### 98.1 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())
```

```
        this.size++;
        this.arr[++top] = e;
        return false;
    }

    public boolean isFull() {
        if (this.size == this.CAP)
            return false;
        return true;
    }

    public String toString() {
        if(this.size==0){
            return null;
        }

        StringBuilder sb = new StringBuilder();
        for(int i=0; i<this.size; i++){
            sb.append(this.arr[i] + ", ");
        }

        sb.setLength(sb.length()-2);
        return sb.toString();
    }

    public static void main(String[] args) {

        Stack<String> stack = new Stack<String>(11);
        stack.push("hello");
        stack.push("world");

        System.out.println(stack);

        stack.pop();
        System.out.println(stack);

        stack.pop();
        System.out.println(stack);
    }
}
```

---

**Output:**

---

```
hello, world
hello
null
```

---

## 98.2 Information from "Effective Java"

It turns out I don't need to improve anything. There are some naming differences but overall my method is ok.

This example occurs twice in "Effective Java". In the first place, the stack example is used to illustrate memory leak. In the second place, the example is used to illustrate when we can suppress unchecked warnings.

Do you wonder how to implement a queue by using an array?

## 99 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

### 99.1 Java Solution

---

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

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

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

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

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

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

        return newHead.next;
    }
}
```

---

What if the digits are stored in regular order instead of reversed order?

Answer: We can simply reverse the list, calculate the result, and reverse the result.

## 100 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.

### 100.1 Analysis

This problem is not straightforward, because it requires "in-place" operations. That means we can only change their pointers, not creating a new list.

### 100.2 Java Solution

This problem can be solved by doing the following:

- Break list in the middle to two lists (use fast & slow pointers)
- Reverse the order of the second list
- Merge two list back together

The following code is a complete runnable class with testing.

---

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

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

public class ReorderList {

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

```
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();
}
}
```

---

### 100.3 Takeaway Messages

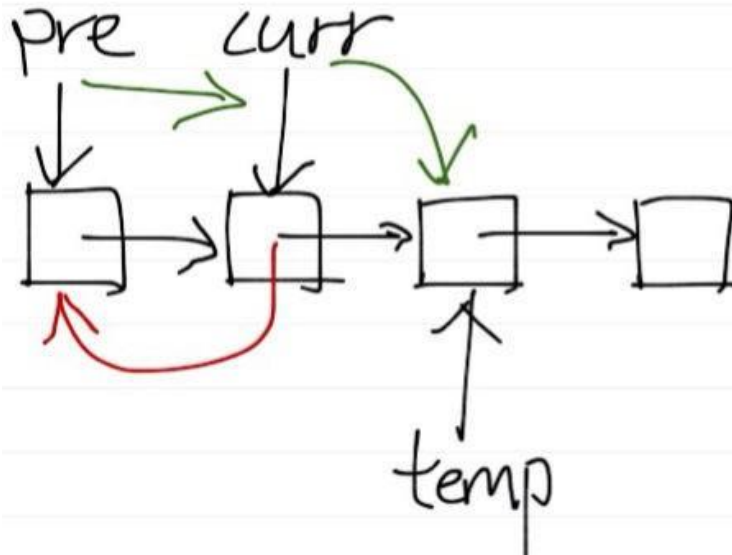
The three steps can be used to solve other problems of linked list. A little diagram may help better understand them.



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

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

head.next = null;
```

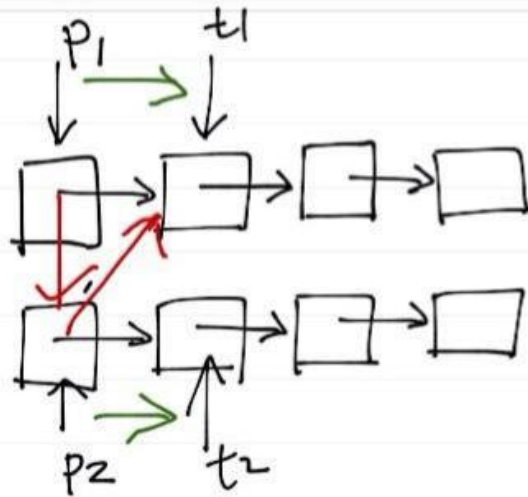


```
ListNode p1 = head;
ListNode p2 = second;

//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;
}
```

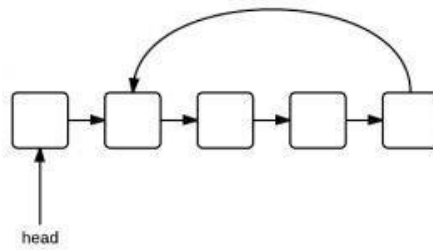


## 101 Linked List Cycle

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

### 101.1 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.



### 101.2 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;
    }
}
```

---

## 102 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.

### 102.1 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) {
```

```
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;
}
```

---

## 102.2 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);
    }
}
```

```
        q.random = null;

        p = p.next;
        q = q.next;
    }

    return newHead;
}
```

---

## 103 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.

### 103.1 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.

### 103.2 Java Solution

---

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

        ListNode p1 = l1;
        ListNode p2 = l2;

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

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

```
        p = p.next;
    }

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

    return fakeHead.next;
}
}
```

---



## 104 Odd Even Linked List

### 104.1 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:

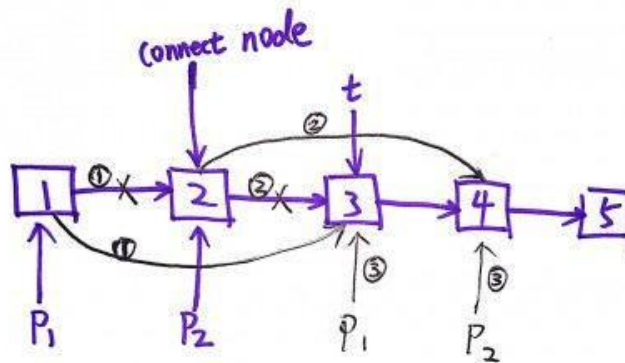
---

```
Given 1->2->3->4->5->NULL,  
return 1->3->5->2->4->NULL.
```

---

### 104.2 Analysis

This problem can be solved by using two pointers. We iterate over the link and move the two pointers.



### 104.3 Java Solution

---

```
public ListNode oddEvenList(ListNode head) {  
    if(head == null)  
        return head;  
  
    ListNode result = head;  
    ListNode p1 = head;  
    ListNode p2 = head.next;
```

```
while(p1 != null && p2 != null){
    ListNode t = p2.next;
    if(t == null)
        break;

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

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

p1.next = connectNode;

return result;
}
```

---

## 105 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.

---

### 105.1 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.

### 105.2 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;
            }
        }
    }
}
```

## 105 Remove Duplicates from Sorted List

---

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

    return head;
}
}
```

---

### 105.3 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;
    }
}
```

---

## 106 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.

### 106.1 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;
}
```

---

## 107 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->4->3->2->5->2 and  $x = 3$ , return 1->2->2->4->3->5.

### 107.1 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;
    }
}
```

## 107 Partition List

---

```
}  
}
```

---

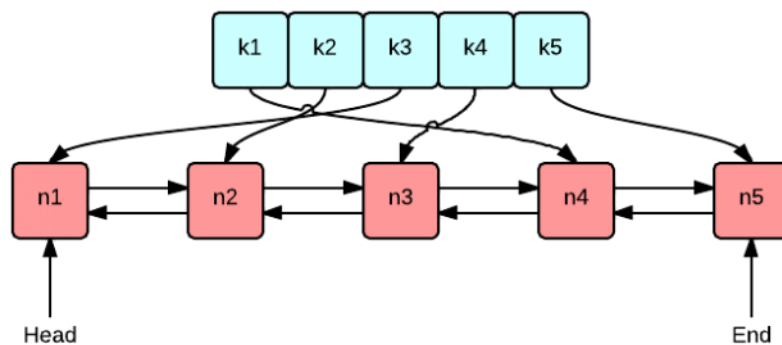
## 108 LRU Cache

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.

### 108.1 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  $O(1)$ .

### 108.2 Java Solution

Define a double linked list node.

```
class Node{
    int key;
    int value;
    Node pre;
    Node next;
```



```
public Node(int key, int value){
    this.key = key;
    this.value = value;
}
}

public class LRUCache {
    int capacity;
    HashMap<Integer, Node> map = new HashMap<Integer, Node>();
    Node head=null;
    Node end=null;

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

    public int get(int key) {
        if(map.containsKey(key)){
            Node n = map.get(key);
            remove(n);
            setHead(n);
            return n.value;
        }

        return -1;
    }

    public void remove(Node n){
        if(n.pre!=null){
            n.pre.next = n.next;
        }else{
            head = n.next;
        }

        if(n.next!=null){
            n.next.pre = n.pre;
        }else{
            end = n.pre;
        }
    }

    public void setHead(Node n){
        n.next = head;
        n.pre = null;

        if(head!=null)
            head.pre = n;
    }
}
```

```
    head = n;

    if (end == null)
        end = head;
}

public void set(int key, int value) {
    if (map.containsKey(key)) {
        Node old = map.get(key);
        old.value = value;
        remove(old);
        setHead(old);
    } else {
        Node created = new Node(key, value);
        if (map.size() >= capacity) {
            map.remove(end.key);
            remove(end);
            setHead(created);
        } else {
            setHead(created);
        }

        map.put(key, created);
    }
}
```

---

## 109 Intersection of Two Linked Lists

### 109.1 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.

### 109.2 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++;
        }
        while(p2 != null){
            len2++;
        }
        int diff = len1 - len2;
        p1 = headA;
        p2 = headB;
        if (diff > 0) {
            for (int i = 0; i < diff; i++)
                p1 = p1.next;
        } else {
            for (int i = 0; i < -diff; i++)
                p2 = p2.next;
        }
        while(p1 != null){
            if (p1 == p2)
                return p1;
            p1 = p1.next;
            p2 = p2.next;
        }
        return null;
    }
}
```

```
    }
    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;
}
}
```

---

## 110 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

---

### 110.1 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;  
}
```

---

## 111 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.

### 111.1 Java Solution

Use two template variable to track the previous and next node of each pair.

---

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

    ListNode h = new ListNode(0);
    h.next = head;
    ListNode p = h;

    while(p.next != null && p.next.next != null){
        //use t1 to track first node
        ListNode t1 = p;
        p = p.next;
        t1.next = p.next;

        //use t2 to track next node of the pair
        ListNode t2 = p.next.next;
        p.next.next = p;
        p.next = t2;
    }

    return h.next;
}
```

---

## 112 Reverse Linked List

Reverse a singly linked list.

### 112.1 Java Solution 1 - Iterative

---

```
public ListNode reverseList(ListNode head) {  
    if(head==null || head.next == null)  
        return head;  
  
    ListNode p1 = head;  
    ListNode p2 = head.next;  
  
    head.next = null;  
    while(p1!= null && p2!= null){  
        ListNode t = p2.next;  
        p2.next = p1;  
        p1 = p2;  
        if (t!=null){  
            p2 = t;  
        }else{  
            break;  
        }  
    }  
  
    return p2;  
}
```

---

### 112.2 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;
```

## 112 Reverse Linked List

---

```
    second.next = head;  
  
    return rest;  
}
```

---



## 113 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.

### 113.1 Analysis

### 113.2 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;
```

```
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;
}
```

---

## 114 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.

### 114.1 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;  
}
```

---

## 114.2 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;
}
```

---

## 115 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.

### 115.1 Analysis

This problem can be solved by using two queues.

### 115.2 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();
}
}
```

---

## 116 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.

### 116.1 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();
    }
}
```

## 116 Implement Queue using Stacks

---

```
public boolean empty() {  
    return value.isEmpty();  
}  
}
```

---



## 117 Palindrome Linked List

Given a singly linked list, determine if it is a palindrome.

### 117.1 Java Solution 1

We can create a new list in reversed order and then compare each node. The time and space are  $O(n)$ .

---

```
public boolean isPalindrome(ListNode head) {  
    if(head == null)  
        return true;  
  
    ListNode p = head;  
    ListNode prev = new ListNode(head.val);  
  
    while(p.next != null){  
        ListNode temp = new ListNode(p.next.val);  
        temp.next = prev;  
        prev = temp;  
        p = p.next;  
    }  
  
    ListNode p1 = head;  
    ListNode p2 = prev;  
  
    while(p1!=null){  
        if(p1.val != p2.val)  
            return false;  
  
        p1 = p1.next;  
        p2 = p2.next;  
    }  
  
    return true;  
}
```

---

### 117.2 Java Solution 2

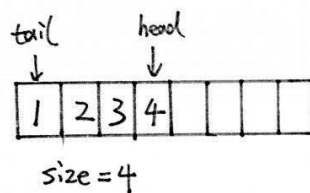
We can use a fast and slow pointer to get the center of the list, then reverse the second

```
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;  
}
```

---

## 118 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);
    }
}
```

---

## 119 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.

### 119.1 Java Solution

---

```
public void deleteNode(ListNode node) {  
    node.val = node.next.val;  
    node.next = node.next.next;  
}
```

---

Is this problem too easy? Or I'm doing it wrong.

## 120 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.

### 120.1 Java Solution

This problem is solved by using a queue.

---

```
public class MovingAverage {

    LinkedList<Integer> queue;
    int size;

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

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

        return (double)sum/queue.size();
    }
}
```

---

