目　录

致谢

当前文档 《Solve Leetcode Problems(英文)》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-04-28。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/solve-leetcode-problems

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

致谢

致谢

Introduction

- Solve Leetcode Problems

Solve Leetcode Problems

This GitBook contains all the LeetCode problems that I have solved.

Donate PayPal

Read my book here:

https://jeantimex.gitbooks.io/solve-leetcode-problems/content/

"The human spirit must prevail over technology.."

- Albert Einstein

Solve leetcode problems

Su

Array

- Array

Array

Find the Celebrity

- Find the Celebrity

Find the Celebrity

Suppose you are at a party with `n` people (labeled from `0` to `n - 1`) and among them, there may exist one celebrity. The definition of a celebrity is that all the other `n - 1` people know him/her but he/she does not know any of them.

Now you want to find out who the celebrity is or verify that there is not one. The only thing you are allowed to do is to ask questions like: "Hi, A. Do you know B?" to get information of whether A knows B. You need to find out the celebrity (or verify there is not one) by asking as few questions as possible (in the asymptotic sense).

You are given a helper function `bool knows(a, b)` which tells you whether A knows B. Implement a function `int findCelebrity(n)` , your function should minimize the number of calls to `knows` .

Note: There will be exactly one celebrity if he/she is in the party. Return the celebrity's label if there is a celebrity in the party. If there is no celebrity, return `-1` .

Solution:

```
1.   /* The knows API is defined in the parent class Relation.
2.        boolean knows(int a, int b); */
3.
4.   public class Solution extends Relation {
5.     public int findCelebrity(int n) {
6.       // base case
7.       if (n <= 0) return -1;
8.       if (n == 1) return 0;
9.
10.      Stack<Integer> stack = new Stack<>();
11.
12.      // put all people to the stack
13.      for (int i = 0; i < n; i++) {
14.        stack.push(i);
15.      }
16.
17.      int a = 0, b = 0;
18.
19.      while (stack.size() > 1) {
20.        a = stack.pop();
21.        b = stack.pop();
22.
23.        if (knows(a, b)) {
24.          // a knows b, so a is not the celebrity, but b may be
25.          stack.push(b);
26.        } else {
27.          // a doesn't know b, so b is not the celebrity, but a may
     be
28.          stack.push(a);
29.        }
30.      }
31.
32.      // double check the potiential celebrity
33.      int c = stack.pop();
34.
35.      for (int i = 0; i < n; i++) {
36.        // c should not know anyone else
37.        if (i != c && (knows(c, i) || !knows(i, c))) {
```

```
38.          return -1;
39.        }
40.      }
41.
42.    return c;
43.    }
44. }
```

First Missing Positive

- First Missing Positive

First Missing Positive

Given an unsorted integer array, find the first missing positive integer.

For example,

Given `[1,2,0]` return `3`, and `[3,4,-1,1]` return `2`.

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

Solution:

```
public class Solution {
  public int firstMissingPositive(int[] a) {
    int i = 0;
    while (i < a.length) {
      if (a[i] > 0 && a[i] <= a.length && a[i] != a[a[i] - 1]) {
        swap(a, i, a[i] - 1);
      } else {
        i++;
      }
    }

    i = 0;
    while (i < a.length && a[i] == i + 1) {
      i++;
    }
```

```
17.        return i + 1;
18.    }
19.
20.    void swap(int[] a, int i, int j) {
21.        int b = a[i];
22.        a[i] = a[j];
23.        a[j] = b;
24.    }
25.  }
```

Game of Life

- Game of Life

Game of Life

According to the Wikipedia's article: "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

Given a board with m by n cells, each cell has an initial state live (1) or dead (0). Each cell interacts with its eight neighbors (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population..
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Write a function to compute the next state (after one update) of the board given its current state.

Follow up:

1. Could you solve it in-place? Remember that the board needs to be updated at the same time: You cannot update some cells first and then use their updated values to update other cells.
2. In this question, we represent the board using a 2D array. In principle, the board is infinite, which would cause problems when the active area encroaches the border of the array. How would you address these problems?

Solution:

```java
public class Solution {
  public void gameOfLife(int[][] board) {
    int n = board.length, m = board[0].length;

    int[] di = {-1, -1, -1,  0, 0,  1, 1, 1};
    int[] dj = {-1,  0,  1, -1, 1, -1, 0, 1};

    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
        int live = 0;

        for (int k = 0; k < 8; k++) {
          int ii = i + di[k], jj = j + dj[k];

          if (ii < 0 || ii >= n || jj < 0 || jj >= m)
            continue;

          if (board[ii][jj] == 1 || board[ii][jj] == 2)
            live++;
        }

        if (board[i][j] == 1 && (live < 2 || live > 3)) {
          board[i][j] = 2;
```

```
24.          } else if (board[i][j] == 0 && live == 3) {
25.            board[i][j] = 3;
26.          }
27.        }
28.      }
29.
30.      for (int i = 0; i < n; i++) {
31.        for (int j = 0; j < m; j++) {
32.          board[i][j] %= 2;
33.        }
34.      }
35.    }
36.  }
```

Majority Element II

- Majority Element II

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.

Solution:

```java
public class Solution {
  public List<Integer> majorityElement(int[] a) {
    // we can only have maximum 2 majority elements
    int n = a.length;
    int c1 = 0, c2 = 0;
    Integer m1 = null, m2 = null;

    // step 1. find out those 2 majority elements
    // using Moore majority voting algorithm
    for (int i = 0; i < n; i++) {
      if (m1 != null && a[i] == m1) {
        c1++;
      } else if (m2 != null && a[i] == m2) {
        c2++;
      } else if (c1 == 0) {
        m1 = a[i]; c1 = 1;
      } else if (c2 == 0) {
        m2 = a[i]; c2 = 1;
      } else {
        c1--; c2--;
      }
    }

    // step 2. double check
```

```
25.     c1 = 0; c2 = 0;
26.
27.     for (int i = 0; i < n; i++) {
28.       if (m1 != null && a[i] == m1) c1++;
29.       if (m2 != null && a[i] == m2) c2++;
30.     }
31.
32.     List<Integer> res = new ArrayList<Integer>();
33.
34.     if (c1 > n / 3) res.add(m1);
35.     if (c2 > n / 3) res.add(m2);
36.
37.     return res;
38.   }
39. }
```

Missing Ranges

- Missing Ranges

Missing Ranges

Given a sorted integer array where the range of elements are [lower, upper] inclusive, return its missing ranges.

For example, given `[0, 1, 3, 50, 75]` , lower = 0 and upper = 99, return `["2", "4->49", "51->74", "76->99"]` .

Solution:

```java
public class Solution {
  public List<String> findMissingRanges(int[] a, int lo, int hi) {
    List<String> res = new ArrayList<String>();

    // the next number we need to find
    int next = lo;

    for (int i = 0; i < a.length; i++) {
      // not within the range yet
      if (a[i] < next) continue;

      // continue to find the next one
      if (a[i] == next) {
        next++;
        continue;
      }

      // get the missing range string format
      res.add(getRange(next, a[i] - 1));

```

```
21.        // now we need to find the next number
22.        next = a[i] + 1;
23.      }
24.
25.      // do a final check
26.      if (next <= hi) res.add(getRange(next, hi));
27.
28.      return res;
29.    }
30.
31.    String getRange(int n1, int n2) {
32.      return (n1 == n2) ? String.valueOf(n1) : String.format("%d-
    >%d", n1, n2);
33.    }
34.  }
```

Next Permutation

- Next Permutation

Next Permutation

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

`1,2,3` → `1,3,2`

`3,2,1` → `1,2,3`

`1,1,5` → `1,5,1`

Solution:

```
1.  public class Solution {
2.    public void nextPermutation(int[] nums) {
3.      if (nums == null || nums.length == 0) {
4.        return;
5.      }
```

```
  6.
  7.      int n = nums.length;
  8.
  9.      // step 1. scan from right and find the digit that is lower
          than the one on its right
 10.      int p = n - 1;
 11.      while (p > 0 && nums[p - 1] >= nums[p]) { p--; }
 12.
 13.      if (p == 0) {
 14.          // no such digit is found, the whole array is sorted in
              descending order
 15.          // we can simply reverse it
 16.          reverse(nums, 0, n - 1);
 17.          return;
 18.      }
 19.
 20.      // step 2. from p, find the digit that is just larger than
          nums[p - 1]
 21.      int i = p - 1;
 22.      int j = p;
 23.
 24.      while (p < n) {
 25.        if (nums[p] > nums[i] && nums[p] <= nums[j]) {
 26.          j = p;
 27.        }
 28.        p++;
 29.      }
 30.
 31.      // step 3. swap i & j
 32.      swap(nums, i, j);
 33.
 34.      // step 4. reverse the digits after i
 35.      reverse(nums, i + 1, n - 1);
 36.    }
 37.
 38.    void swap(int[] nums, int i, int j) {
 39.      int t = nums[i];
 40.      nums[i] = nums[j];
```

```
41.      nums[j] = t;
42.    }
43.
44.    void reverse(int[] nums, int i, int j) {
45.      while (i < j) {
46.        swap(nums, i++, j--);
47.      }
48.    }
49.  }
```

Pascal's Triangle

- Pascal's Triangle

Pascal's Triangle

Given numRows, generate the first numRows of Pascal's triangle.

For example, given numRows = 5,
Return

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

Solution:

```
1.  public class Solution {
2.    public List<List<Integer>> generate(int numRows) {
3.      List<List<Integer>> res = new ArrayList<List<Integer>>();
4.
5.      for (int i = 0; i < numRows; i++) {
6.        List<Integer> list = new ArrayList<Integer>
    (Arrays.asList(1));
7.
8.        for (int j = 1; j < i; j++) {
9.          list.add(res.get(i - 1).get(j - 1) + res.get(i -
    1).get(j));
10.       }
11.
```

```
12.         if (i > 0) list.add(1);
13.
14.         res.add(list);
15.     }
16.
17.     return res;
18.   }
19. }
```

Pascal's Triangle II

- Pascal's Triangle II

Pascal's Triangle II

Given an index k, return the kth row of the Pascal's triangle.

For example, given k = 3,

Return `[1,3,3,1]` .

Note:

Could you optimize your algorithm to use only O(k) extra space?

Solution:

```
1.  public class Solution {
2.    public List<Integer> getRow(int k) {
3.      Integer[] arr = new Integer[k + 1];
4.      Arrays.fill(arr, 0);
5.      arr[0] = 1;
6.
7.      for (int i = 1; i <= k; i++) {
8.        for (int j = i; j > 0; j--) {
9.          arr[j] = arr[j] + arr[j - 1];
10.        }
11.      }
12.
13.      return Arrays.asList(arr);
14.    }
15.  }
```

Product of Array Except Self

- Product of Array Except Self

Product of Array Except Self

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

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

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

Follow up:

Could you solve it with constant space complexity? (Note: The output array does not count as extra space for the purpose of space complexity analysis.)

Solution:

```
1.  public class Solution {
2.    public int[] productExceptSelf(int[] nums) {
3.      int n = nums.length;
4.
5.      int[] left = new int[n];
6.      left[0] = 1;
7.
8.      for (int i = 1; i < n; i++) {
9.        left[i] = nums[i - 1] * left[i - 1];
10.     }
11.
```

```
12.        int[] right = new int[n];
13.        right[n - 1] = 1;
14.
15.        for (int i = n - 2; i >= 0; i--) {
16.          right[i] = nums[i + 1] * right[i + 1];
17.        }
18.
19.        int[] res = new int[n];
20.        for (int i = 0; i < n; i++) {
21.          res[i] = left[i] * right[i];
22.        }
23.
24.        return res;
25.      }
26.  }
```

Range Addition

Range Addition

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

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

Return the modified array after all **k** operations were executed.

Example:

```
 1.  Given:
 2.    length = 5,
 3.    updates = [
 4.      [1,   3,   2],
 5.      [2,   4,   3],
 6.      [0,   2, -2]
 7.    ]
 8.
 9.  Output:
10.    [-2, 0, 3, 5, 3]
```

Explanation:

```
 1.  Initial state:
 2.  [ 0, 0, 0, 0, 0 ]
```

```
 3.
 4.  After applying operation [1, 3, 2]:
 5.  [ 0, 2, 2, 2, 0 ]
 6.
 7.  After applying operation [2, 4, 3]:
 8.  [ 0, 2, 5, 5, 3 ]
 9.
10.  After applying operation [0, 2, -2]:
11.  [-2, 0, 3, 5, 3 ]
```

Hint:

1. Thinking of using advanced data structures? You are thinking it too complicated.
2. For each update operation, do you really need to update all elements between i and j?
3. Update only the first and end element is sufficient.
4. The optimal time complexity is O(k + n) and uses O(1) extra space.

Rotate Array

- Rotate Array

Rotate Array

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]` .

Note:

Try to come up as many solutions as you can, there are at least 3 different ways to solve this problem.

Hint:

Could you do it in-place with O(1) extra space? Related problem: Reverse Words in a String II

Solution:

```java
public class Solution {
  public void rotate(int[] nums, int k) {
    int n = nums.length;

    k %= n;

    reverse(nums, 0, n - k - 1);
    reverse(nums, n - k, n - 1);
    reverse(nums, 0, n - 1);
  }

```

```
12.    private void reverse(int[] nums, int i, int j) {
13.      while (i < j) {
14.        int val = nums[i];
15.        nums[i++] = nums[j];
16.        nums[j--] = val;
17.      }
18.    }
19.  }
```

Rotate Image

- Rotate Image

Rotate Image

You are given an n x n 2D matrix representing an image.

Rotate the image by 90 degrees (clockwise).

Follow up:

Could you do this in-place?

Solution:

```java
public class Solution {
  public void rotate(int[][] matrix) {
    if (matrix == null) {
      return;
    }

    int n = matrix.length;
    int layers = n / 2;

    for (int k = 0; k < layers; k++) {
      for (int i = 0; i < n - 1; i++) {
        int tmp = matrix[k][k + i];

        // left to top
        matrix[k][k + i] = matrix[k + n - 1 - i][k];

        // bottom to left
        matrix[k + n - 1 - i][k] = matrix[k + n - 1][k + n - 1 -
    i];
```

```
19.
20.          // right to bottom
21.          matrix[k + n - 1][k + n - 1 - i] = matrix[k + i][k + n -
     1];
22.
23.          // top to right
24.          matrix[k + i][k + n - 1] = tmp;
25.        }
26.
27.        n -= 2;
28.      }
29.    }
30. }
```

Set Matrix Zeroes

- Set Matrix Zeroes

Set Matrix Zeroes

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

Follow up:

Did you use extra space?

A straight forward solution using O(mn) space is probably a bad idea.

A simple improvement uses O(m + n) space, but still not the best solution.

Could you devise a constant space solution?

Solution:

```java
public class Solution {
  public void setZeroes(int[][] matrix) {
    int n = matrix.length;
    int m = matrix[0].length;

    // Check if we need to set zeros for first row and column
    boolean cleanFirstRow = false;
    boolean cleanFirstCol = false;

    for (int j = 0; j < m; j++) {
      if (matrix[0][j] == 0) {
        cleanFirstRow = true;
```

```
13.            break;
14.          }
15.        }
16.
17.        for (int i = 0; i < n; i++) {
18.          if (matrix[i][0] == 0) {
19.            cleanFirstCol = true;
20.            break;
21.          }
22.        }
23.
24.        // Use first row and first column to save the flags
25.        for (int i = 1; i < n; i++) {
26.          for (int j = 1; j < m; j++) {
27.            if (matrix[i][j] == 0) {
28.              matrix[0][j] = 0;
29.              matrix[i][0] = 0;
30.            }
31.          }
32.        }
33.
34.        // Based on the flags set other cells
35.        for (int i = 1; i < n; i++) {
36.          for (int j = 1; j < m; j++) {
37.            if (matrix[0][j] == 0 || matrix[i][0] == 0) {
38.              matrix[i][j] = 0;
39.            }
40.          }
41.        }
42.
43.        // At last set the first row and column
44.        if (cleanFirstRow) {
45.          for (int j = 0; j < m; j++) {
46.            matrix[0][j] = 0;
47.          }
48.        }
49.
50.        if (cleanFirstCol) {
```

```
51.        for (int i = 0; i < n; i++) {
52.          matrix[i][0] = 0;
53.        }
54.      }
55.    }
56. }
```

Shortest Word Distance

- Shortest Word Distance

Shortest Word Distance

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

For example,

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

Given word1 = `"coding"` , word2 = `"practice"` , return 3.
Given word1 = `"makes"` , word2 = `"coding"` , return 1.

Note:

You may assume that word1 **does not equal to** word2, and word1 and word2 are both in the list.

Solution:

```
1.  public class Solution {
2.    public int shortestDistance(String[] words, String word1, String
    word2) {
3.      int p1 = -1, p2 = -1, min = words.length;
4.      boolean updated = false;
5.
6.      for (int i = 0; i < words.length; i++) {
7.        if (word1.equals(words[i])) {
```

```
 8.            p1 = i;
 9.            updated = true;
10.         }
11.
12.        if (word2.equals(words[i])) {
13.          p2 = i;
14.          updated = true;
15.        }
16.
17.        if (p1 != -1 && p2 != -1 && updated) {
18.          updated = false;
19.          min = Math.min(min, Math.abs(p1 - p2));
20.        }
21.      }
22.
23.      return min;
24.    }
25. }
```

Shortest Word Distance III

Shortest Word Distance III

This is a follow up of Shortest Word Distance. The only difference is now word1 could be the same as word2.

Given a list of words and two words word1 and word2, return the shortest distance between these two words in the list.

word1 and word2 may be the same and they represent two individual words in the list.

For example,

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

Given word1 = `"makes"` , word2 = `"coding"` , return 1.
Given word1 = `"makes"` , word2 = `"makes"` , return 3.

Note:

You may assume word1 and word2 are both in the list.

Solution:

```
1.  public class Solution {
2.    public int shortestWordDistance(String[] words, String word1,
      String word2) {
```

```
 3.      if (word1.equals(word2))
 4.        return helper1(words, word1);
 5.      else
 6.        return helper2(words, word1, word2);
 7.    }
 8.
 9.    int helper1(String[] words, String word) {
10.      int p = -1, min = Integer.MAX_VALUE;
11.
12.      for (int i = 0; i < words.length; i++) {
13.        if (words[i].equals(word)) {
14.          if (p != -1) {
15.            min = Math.min(min, i - p);
16.          }
17.          p = i;
18.        }
19.      }
20.
21.      return min;
22.    }
23.
24.    int helper2(String[] words, String word1, String word2) {
25.      int p1 = -1, p2 = -1, min = Integer.MAX_VALUE;
26.
27.      for (int i = 0; i < words.length; i++) {
28.        if (words[i].equals(word1)) {
29.          p1 = i;
30.        }
31.
32.        if (words[i].equals(word2)) {
33.          p2 = i;
34.        }
35.
36.        if (p1 != -1 && p2 != -1) {
37.          min = Math.min(min, Math.abs(p1 - p2));
38.        }
39.      }
40.
```

```
41.        return min;
42.    }
43. }
```

Spiral Matrix

- Spiral Matrix

Spiral Matrix

Given a matrix of m x n elements (m rows, n columns), return all elements of the matrix in spiral order.

For example,
Given the following matrix:

```
1.  [
2.   [ 1, 2, 3 ],
3.   [ 4, 5, 6 ],
4.   [ 7, 8, 9 ]
5.  ]
```

You should return `[1,2,3,6,9,8,7,4,5]` .

Solution:

```
1.  public class Solution {
2.    public List<Integer> spiralOrder(int[][] matrix) {
3.      List<Integer> res = new ArrayList<Integer>();
4.
5.      if (matrix == null || matrix.length == 0 || matrix[0].length ==
   0) {
6.        return res;
7.      }
8.
9.      int m = matrix.length;
10.     int n = matrix[0].length;
11.
```

```
12.        int layers = (int)Math.ceil(Math.min(m, n) / 2.0);
13.
14.      for (int k = 0; k < layers; k++) {
15.        if (m <= 0 || n <= 0) {
16.          break;
17.        }
18.
19.        // one row
20.        if (m == 1) {
21.          for (int j = 0; j < n; j++) {
22.            res.add(matrix[k][k + j]);
23.          }
24.        }
25.
26.        // one column
27.        else if (n == 1) {
28.          for (int i = 0; i < m; i++) {
29.            res.add(matrix[k + i][k]);
30.          }
31.        }
32.
33.        else {
34.          // top
35.          for (int j = 0; j < n - 1; j++) {
36.            res.add(matrix[k][k + j]);
37.          }
38.
39.          // right
40.          for (int i = 0; i < m - 1; i++) {
41.            res.add(matrix[k + i][k + n - 1]);
42.          }
43.
44.          // bottom
45.          for (int j = n - 1; j > 0; j--) {
46.            res.add(matrix[k + m - 1][k + j]);
47.          }
48.
49.          // left
```

```
50.          for (int i = m - 1; i > 0; i--) {
51.              res.add(matrix[k + i][k]);
52.          }
53.       }
54.
55.       m -= 2;
56.       n -= 2;
57.     }
58.
59.     return res;
60.   }
61. }
```

Spiral Matrix II

Spiral Matrix II

Given an integer n, generate a square matrix filled with elements from 1 to n2 in spiral order.

For example,
Given n = 3,

You should return the following matrix:

```
1.  [
2.   [ 1, 2, 3 ],
3.   [ 8, 9, 4 ],
4.   [ 7, 6, 5 ]
5.  ]
```

Solution:

```
1.  public class Solution {
2.    public int[][] generateMatrix(int n) {
3.      int[][] res = new int[n][n];
4.
5.      int layers = (int)Math.ceil(n / 2.0);
6.
7.      int val = 1;
8.
9.      for (int k = 0; k < layers; k++) {
10.       if (n == 1) {
11.         res[k][k] = val++;
12.         break;
13.       }
```

```
14.
15.        // top
16.        for (int j = 0; j < n - 1; j++) {
17.          res[k][k + j] = val++;
18.        }
19.
20.        // right
21.        for (int i = 0; i < n - 1; i++) {
22.          res[k + i][k + n - 1] = val++;
23.        }
24.
25.        // bottom
26.        for (int j = n - 1; j > 0; j--) {
27.          res[k + n - 1][k + j] = val++;
28.        }
29.
30.        // left
31.        for (int i = n - 1; i > 0; i--) {
32.          res[k + i][k] = val++;
33.        }
34.
35.        n -= 2;
36.      }
37.
38.      return res;
39.    }
40.  }
```

Summary Ranges

- Summary Ranges

Summary Ranges

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

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

Solution:

```
1.  public class Solution {
2.    public List<String> summaryRanges(int[] a) {
3.      List<String> res = new ArrayList<String>();
4.
5.      if (a == null)
6.        return res;
7.
8.      int i = 0, n = a.length;
9.
10.     for (int j = 1; j < n; j++) {
11.       if ((long)a[j] - (long)a[j - 1] > 1) {
12.         // found a range
13.         res.add(getRange(a[i], a[j - 1]));
14.         i = j;
15.       }
16.     }
17.
18.     // do a final check
19.     if (i < n)
20.       res.add(getRange(a[i], a[n - 1]));
21.
22.     return res;
```

```
23.      }
24.
25.    String getRange(int n1, int n2) {
26.      return (n1 == n2) ? String.valueOf(n1) : String.format("%d-
     >%d", n1, n2);
27.    }
28.  }
```

Backtracking

- Backtracking

Backtracking

Combination Sum II

- Combination Sum II

Combination Sum II

Given a collection of candidate numbers (**C**) and a target number (**T**), find all unique combinations in C where the candidate numbers sums to **T**.

Each number in **C** may only be used **once** in the combination.

Note:

- All numbers (including target) will be positive integers.
- The solution set must not contain duplicate combinations.

For example, given candidate set `[10, 1, 2, 7, 6, 1, 5]` and target `8`, A solution set is:

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

Solution:

```java
public class Solution {
  public List<List<Integer>> combinationSum2(int[] candidates, int
target) {
    Arrays.sort(candidates);
    List<List<Integer>> res = new ArrayList<List<Integer>>();
    helper(candidates, target, 0, new ArrayList<Integer>(), res);
    return res;
  }

  private void helper(int[] candidates, int target, int index,
List<Integer> sol, List<List<Integer>> res) {
    if (target == 0) {
      res.add(new ArrayList<Integer>(sol));
      return;
    }

    if (target < 0 || index == candidates.length) {
      return;
    }

    for (int i = index; i < candidates.length; i++) {
      if (i > index && candidates[i] == candidates[i - 1]) {
        continue;
      }

      sol.add(candidates[i]);
      helper(candidates, target - candidates[i], i + 1, sol, res);
      sol.remove(sol.size() - 1);
    }
  }
}
```

Combination Sum III

- Combination Sum III

Combination Sum III

Find all possible combinations of **k** numbers that add up to a number **n**, given that only numbers from 1 to 9 can be used and each combination should be a unique set of numbers.

Example 1:

Input: **k** = 3, **n** = 7

Output:

```
1.  [[1,2,4]]
```

Example 2:

Input: **k** = 3, **n** = 9

Output:

```
1.  [[1,2,6], [1,3,5], [2,3,4]]
```

Solution:

```
1.  public class Solution {
2.    public List<List<Integer>> combinationSum3(int k, int n) {
3.      List<List<Integer>> res = new ArrayList<List<Integer>>();
4.      helper(k, n, 1, new ArrayList<Integer>(), res);
5.      return res;
```

```
 6.     }
 7.
 8.     private void helper(int k, int n, int index, List<Integer> sol,
        List<List<Integer>> res) {
 9.         if (n == 0 && k == sol.size()) {
10.             res.add(new ArrayList<Integer>(sol));
11.             return;
12.         }
13.
14.         if (index > 9 || n < 0) {
15.             return;
16.         }
17.
18.         for (int i = index; i <= 9; i++) {
19.             sol.add(i);
20.             helper(k, n - i, i + 1, sol, res);
21.             sol.remove(sol.size() - 1);
22.         }
23.     }
24. }
```

Combination Sum

- Combination Sum

Combination Sum

Given a set of candidate numbers (**C**) and a target number (**T**), find all unique combinations in **C** where the candidate numbers sums to **T**.

The **same** repeated number may be chosen from **C** unlimited number of times.

Note:

- All numbers (including target) will be positive integers.
- The solution set must not contain duplicate combinations.

For example, given candidate set `[2, 3, 6, 7]` and target `7`,
A solution set is:

```
1.  [
2.    [7],
3.    [2, 2, 3]
4.  ]
```

Solution:

```
1.  public class Solution {
2.    public List<List<Integer>> combinationSum(int[] candidates, int
```

```
          target) {
 3.           Arrays.sort(candidates);
 4.           List<List<Integer>> res = new ArrayList<List<Integer>>();
 5.           helper(candidates, target, 0, new ArrayList<Integer>(), res);
 6.           return res;
 7.       }
 8.
 9.     private void helper(int[] candidates, int target, int index,
          List<Integer> sol, List<List<Integer>> res) {
10.         if (target == 0) {
11.             res.add(new ArrayList<Integer>(sol));
12.             return;
13.         }
14.
15.         if (target < 0 || index == candidates.length) {
16.             return;
17.         }
18.
19.         for (int i = index; i < candidates.length; i++) {
20.             sol.add(candidates[i]);
21.             helper(candidates, target - candidates[i], i, sol, res);
22.             sol.remove(sol.size() - 1);
23.         }
24.     }
25. }
```

Combinations

- Combinations

Combinations

Given two integers n and k, return all possible combinations of k numbers out of 1 … n.

For example,

If n = 4 and k = 2, a solution is:

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

Solution:

```
1.  public class Solution {
2.    public List<List<Integer>> combine(int n, int k) {
3.      List<List<Integer>> res = new ArrayList<>();
4.      List<Integer> sol = new ArrayList<>();
5.
6.      helper(n, k, 1, sol, res);
7.
8.      return res;
9.    }
10.
11.    void helper(int n, int k, int m, List<Integer> sol,
```

```
       List<List<Integer>> res) {
12.        if (sol.size() == k) {
13.            res.add(new ArrayList<Integer>(sol));
14.            return;
15.        }
16.
17.        for (int i = m; i <= n; i++) {
18.            sol.add(i);
19.            helper(n, k, i + 1, sol, res);
20.            sol.remove(sol.size() - 1);
21.        }
22.    }
23. }
```

Factor Combinations

- Factor Combinations

Factor Combinations

Numbers can be regarded as product of its factors. For example,

```
1.  8 = 2 x 2 x 2;
2.    = 2 x 4.
```

Write a function that takes an integer n and return all possible combinations of its factors.

Note:

1. You may assume that n is always positive.
2. Factors should be greater than 1 and less than n.

Examples:

input: `1`

output:

```
1. []
```

input: `37`

output:

```
1. []
```

input: `12`

output:

```
1.  [
2.    [2, 6],
3.    [2, 2, 3],
4.    [3, 4]
5.  ]
```

input: `32`

output:

```
1.  [
2.    [2, 16],
3.    [2, 2, 8],
4.    [2, 2, 2, 4],
5.    [2, 2, 2, 2, 2],
6.    [2, 4, 4],
7.    [4, 8]
8.  ]
```

Flip Game II

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 `"+--+"` .

Follow up:

Derive your algorithm's runtime complexity.

Solution:

```
1.  public class Solution {
2.    public boolean canWin(String s) {
3.      if (s == null || s.length() < 2) return false;
4.
5.      Map<String, Boolean> map = new HashMap<>();
6.      return canWin(s, map);
7.    }
```

```
 8.
 9.    public boolean canWin(String s, Map<String, Boolean> map){
10.      if (map.containsKey(s)) return map.get(s);
11.
12.      for (int i = 0; i < s.length() - 1; i++) {
13.        if (s.charAt(i) == '+' && s.charAt(i + 1) == '+') {
14.
15.          String sOpponent = s.substring(0, i) + "--" + s.substring(i
   + 2);
16.          if (!canWin(sOpponent, map)) {
17.            map.put(s, true);
18.            return true;
19.          }
20.        }
21.      }
22.
23.      map.put(s, false);
24.      return false;
25.    }
26. }
```

Gray Code

- Gray Code

Gray Code

The gray code is a binary numeral system where two successive values differ in only one bit.

Given a non-negative integer n representing the total number of bits in the code, print the sequence of gray code. A gray code sequence must begin with 0.

For example, given n = 2, return `[0,1,3,2]` . Its gray code sequence is:

```
1.  00 - 0
2.  01 - 1
3.  11 - 3
4.  10 - 2
```

Note:

For a given n, a gray code sequence is not uniquely defined.

For example, `[0,2,3,1]` is also a valid gray code sequence according to the above definition.

For now, the judge is able to judge based on one instance of gray code sequence. Sorry about that.

Solution

```java
public class Solution {
  public List<Integer> grayCode(int n) {
    List<Integer> res = new ArrayList<Integer>();

    if (n == 0) {
      res.add(0);
      return res;
    }

    if (n == 1) {
      res.add(0);
      res.add(1);
      return res;
    }

    res.add(0);
    res.add(1);

    for (int k = 2; k <= n; k++) {
      int size = res.size();

      for (int i = size - 1; i >= 0; i--) {
        int msb = 1 << (k - 1);
        int code = res.get(i) + msb;
        res.add(code);
      }
    }

    return res;
  }
}
```

N-Queens

- N-Queens

N-Queens

The n-queens puzzle is the problem of placing n queens on an n×n chessboard such that no two queens attack each other.

One solution to the eight queens puzzle
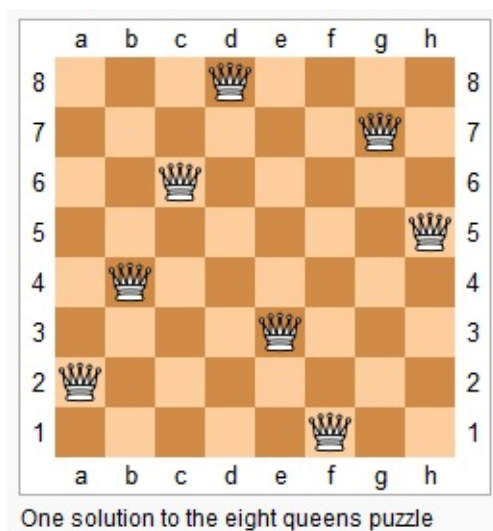
Given an integer n, return all distinct solutions to the n-queens puzzle.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space respectively.

For example,

There exist two distinct solutions to the 4-queens puzzle:

```
 1.  [
 2.   [".Q..",  // Solution 1
 3.    "...Q",
 4.    "Q...",
 5.    "..Q."],
 6.
 7.   ["..Q.",  // Solution 2
 8.    "Q...",
 9.    "...Q",
10.    ".Q.."]
11.  ]
```

Solution:

```
 1.  public class Solution {
 2.    public List<String[]> solveNQueens(int n) {
 3.      List<String[]> res = new ArrayList<String[]>();
 4.      helper(n, 0, new int[n], res);
 5.      return res;
 6.    }
 7.
 8.    private void helper(int n, int row, int[] columnForRow,
    List<String[]> res) {
 9.      if (row == n) {
10.        String[] sol = new String[n];
11.        for (int r = 0; r < n; r++) {
12.          sol[r] = "";
13.          for (int c = 0; c < n; c++) {
14.            sol[r] += (columnForRow[r] == c) ? "Q" : ".";
15.          }
16.        }
17.        res.add(sol);
18.        return;
19.      }
20.
21.      for (int col = 0; col < n; col++) {
22.        columnForRow[row] = col;
```

```
23.
24.        if (check(row, col, columnForRow)) {
25.          helper(n, row + 1, columnForRow, res);
26.        }
27.      }
28.    }
29.
30.    private boolean check(int row, int col, int[] columnForRow) {
31.      for (int r = 0; r < row; r++) {
32.        if (columnForRow[r] == col || row - r ==
    Math.abs(columnForRow[r] - col)) {
33.          return false;
34.        }
35.      }
36.      return true;
37.    }
38. }
```

N-Queens II

- N-Queens II

N-Queens II

Follow up for N-Queens problem.

Now, instead outputting board configurations, return the total number of distinct solutions.

One solution to the eight queens puzzle
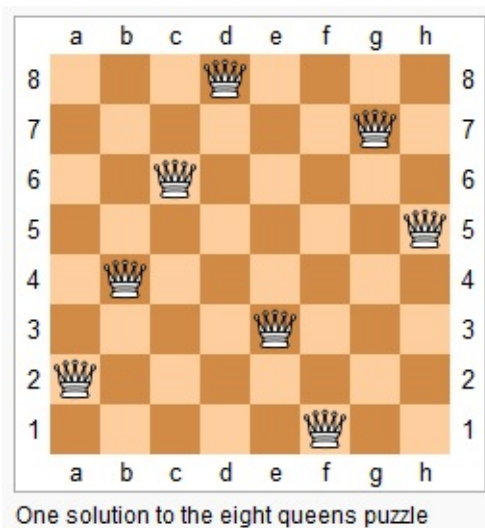
Solution:

```
1.  public class Solution {
2.    int count;
3.
4.    public int totalNQueens(int n) {
5.      count = 0;
6.      helper(n, 0, new int[n]);
7.      return count;
8.    }
9.
10.   private void helper(int n, int row, int[] columnForRow) {
11.     if (row == n) {
```

```
12.        count++;
13.        return;
14.      }
15.
16.      for (int col = 0; col < n; col++) {
17.        columnForRow[row] = col;
18.        if (check(row, col, columnForRow)) {
19.          helper(n, row + 1, columnForRow);
20.        }
21.      }
22.    }
23.
24.    private boolean check(int row, int col, int[] columnForRow) {
25.      for (int r = 0; r < row; r++) {
26.        if (columnForRow[r] == col || row - r ==
     Math.abs(columnForRow[r] - col)) {
27.          return false;
28.        }
29.      }
30.      return true;
31.    }
32.  }
```

Palindrome Partitioning

- Palindrome Partitioning

Palindrome Partitioning

Given a string s, partition s such that every substring of the partition is a palindrome.

Return all possible palindrome partitioning of s.

For example, given s = `"aab"`,

Return

```
1.  [
2.    ["aa","b"],
3.    ["a","a","b"]
4.  ]
```

Solution:

```
1.  public class Solution {
2.    public List<List<String>> partition(String s) {
3.      List<List<String>> res = new ArrayList<List<String>>();
4.
5.      if (s == null || s.length() == 0) {
6.        return res;
7.      }
8.
9.      int n = s.length();
10.
11.      // step 1. build the dp matrix to hold the palindrome
      information
12.      // dp[i][j] represents whether s[i] to s[j] can form a
```

```
    palindrome
13.    boolean[][] dp = buildMatrix(s, n);
14.
15.    // step 2. recursively generate the list
16.    helper(s, dp, 0, n, new ArrayList<String>(), res);
17.
18.    return res;
19.  }
20.
21.  void helper(String s, boolean[][] dp, int start, int end,
    List<String> sol, List<List<String>> res) {
22.    if (start == end) {
23.      res.add(new ArrayList<String>(sol));
24.      return;
25.    }
26.
27.    for (int i = start; i < end; i++) {
28.      if (dp[start][i]) {
29.        // s[start] to s[i] is a palindrome
30.        sol.add(s.substring(start, i + 1));
31.        helper(s, dp, i + 1, end, sol, res);
32.        sol.remove(sol.size() - 1);
33.      }
34.    }
35.  }
36.
37.  boolean[][] buildMatrix(String s, int n) {
38.    boolean[][] dp = new boolean[n][n];
39.
40.    for (int i = n - 1; i >= 0; i--) {
41.      for (int j = i; j < n; j++) {
42.        if (s.charAt(i) == s.charAt(j) && (j - i <= 2 || dp[i + 1]
    [j - 1])) {
43.          dp[i][j] = true;
44.        }
45.      }
46.    }
47.
```

```
48.        return dp;
49.    }
50. }
```

Palindrome Permutation II

- Palindrome Permutation II

Palindrome Permutation II

Given a string s, return all the palindromic permutations (without duplicates) of it. Return an empty list if no palindromic permutation could be form.

For example:

Given `s = "aabb"`, return `["abba", "baab"]`.

Given `s = "abc"`, return `[]`.

Hint:

1. If a palindromic permutation exists, we just need to generate the first half of the string.
2. To generate all distinct permutations of a (half of) string, use a similar approach from: Permutations II or Next Permutation.

Solution:

```
1.  public class Solution {
2.    public List<String> generatePalindromes(String s) {
3.      int odd = 0;
4.      String mid = "";
5.      List<String> res = new ArrayList<>();
6.      List<Character> list = new ArrayList<>();
7.      Map<Character, Integer> map = new HashMap<>();
```

```
 8.
 9.        // step 1. build character count map and count odds
10.        for (int i = 0; i < s.length(); i++) {
11.          char c = s.charAt(i);
12.          map.put(c, map.containsKey(c) ? map.get(c) + 1 : 1);
13.          odd += map.get(c) % 2 != 0 ? 1 : -1;
14.        }
15.
16.        // cannot form any palindromic string
17.        if (odd > 1) return res;
18.
19.        // step 2. add half count of each character to list
20.        for (Map.Entry<Character, Integer> entry : map.entrySet()) {
21.          char key = entry.getKey();
22.          int val = entry.getValue();
23.
24.          if (val % 2 != 0) {
25.            mid += key;
26.          }
27.
28.          for (int i = 0; i < val / 2; i++) list.add(key);
29.        }
30.
31.        // step 3. generate all the permutations O(n!/2)
32.        getPerm(list, mid, new boolean[list.size()], new
    StringBuilder(), res);
33.
34.        return res;
35.      }
36.
37.      // generate all unique permutation from list
38.      void getPerm(List<Character> list, String mid, boolean[] used,
    StringBuilder sb, List<String> res) {
39.        if (sb.length() == list.size()) {
40.          // form the palindromic string
41.          res.add(sb.toString() + mid + sb.reverse().toString());
42.          sb.reverse();
43.          return;
```

```
44.       }
45.
46.       for (int i = 0; i < list.size(); i++) {
47.         // avoid duplication
48.         if (i > 0 && list.get(i) == list.get(i - 1) && !used[i - 1])
    {
49.           continue;
50.         }
51.
52.         if (!used[i]) {
53.           used[i] = true;
54.           sb.append(list.get(i));
55.
56.           getPerm(list, mid, used, sb, res);
57.
58.           sb.deleteCharAt(sb.length() - 1);
59.           used[i] = false;
60.         }
61.       }
62.     }
63.  }
```

Permutations

- Permutations

Permutations

Given a collection of distinct numbers, return all possible permutations.

For example,

`[1,2,3]` have the following permutations:

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

Solution:

```java
1.  public class Solution {
2.    public List<List<Integer>> permute(int[] nums) {
3.      List<List<Integer>> res = new ArrayList<>();
4.
5.      if (nums.length == 0)
6.        return res;
7.
8.      res.add(new ArrayList<Integer>(Arrays.asList(nums[0])));
9.
10.     for (int i = 1; i< nums.length; i++) {
11.       List<List<Integer>> next = new ArrayList<>();
```

```
12.
13.        for (int j = 0; j <= i; j++) {
14.          for (List<Integer> list : res) {
15.            // make a copy of existing perm
16.            List<Integer> copy = new ArrayList<>(list);
17.            copy.add(j, nums[i]);
18.            next.add(copy);
19.          }
20.        }
21.
22.        res = next;
23.      }
24.
25.      return res;
26.    }
27. }
```

Permutations II

- Permutations II

Permutations II

Given a collection of numbers that might contain duplicates, return all possible unique permutations.

For example,

`[1,1,2]` have the following unique permutations:

```
1.  [
2.    [1,1,2],
3.    [1,2,1],
4.    [2,1,1]
5.  ]
```

Solution:

```
1.  public class Solution {
2.    public List<List<Integer>> permuteUnique(int[] nums) {
3.      Arrays.sort(nums);
4.      List<List<Integer>> res = new ArrayList<List<Integer>>();
5.      boolean[] used = new boolean[nums.length];
6.      helper(nums, used, new ArrayList<Integer>(), res);
7.      return res;
8.    }
9.
10.   private void helper(int[] nums, boolean[] used, List<Integer> sol, List<List<Integer>> res) {
11.     if (sol.size() == nums.length) {
12.       res.add(new ArrayList<Integer>(sol));
13.       return;
```

```
14.        }
15.
16.        for (int i = 0; i < nums.length; i++) {
17.          // avoid duplication
18.          if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
19.            continue;
20.          }
21.
22.          if (!used[i]) {
23.            used[i] = true;
24.            sol.add(nums[i]);
25.            helper(nums, used, sol, res);
26.            sol.remove(sol.size() - 1);
27.            used[i] = false;
28.          }
29.        }
30.      }
31.  }
```

Subsets II

Subsets II

Given a collection of integers that might contain duplicates, nums, return all possible subsets.

Note: The solution set must not contain duplicate subsets.

For example,

If **nums** = `[1,2,2]` , a solution is:

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

Solution:

```
1.  public class Solution {
2.    public List<List<Integer>> subsetsWithDup(int[] nums) {
3.      List<List<Integer>> res = new ArrayList<List<Integer>>();
4.      res.add(new ArrayList<Integer>());
5.
6.      Arrays.sort(nums);
7.
8.      int lastSize = 0;
```

```
 9.
10.      for (int i = 0; i < nums.length; i++) {
11.          int size = res.size();
12.          int start = (i == 0 || nums[i] != nums[i - 1]) ? 0 :
     lastSize;
13.
14.          for (int j = start; j < size; j++) {
15.            List<Integer> sol = new ArrayList<Integer>(res.get(j));
16.            sol.add(nums[i]);
17.            res.add(sol);
18.          }
19.
20.          lastSize = size;
21.      }
22.
23.      return res;
24.    }
25.  }
```

Word Pattern II

- Word Pattern II

Word Pattern II

Given a `pattern` and a string `str`, find if `str` follows the same pattern.

Here follow means a full match, such that there is a bijection between a letter in `pattern` and a non-empty substring in `str`.

Examples:

1. pattern = `"abab"`, str = `"redblueredblue"` should return true.
2. pattern = `"aaaa"`, str = `"asdasdasdasd"` should return true.
3. pattern = `"aabb"`, str = `"xyzabcxzyabc"` should return false.

Notes:

You may assume both `pattern` and `str` contains only lowercase letters.

Solution:

```
1.  public class Solution {
2.    public boolean wordPatternMatch(String pattern, String str) {
3.      Map<Character, String> map = new HashMap<>();
4.      Set<String> set = new HashSet<>();
5.
```

```
 6.        return isMatch(str, 0, pattern, 0, map, set);
 7.     }
 8.
 9.     boolean isMatch(String str, int i, String pat, int j,
     Map<Character, String> map, Set<String> set) {
10.        // base case
11.        if (i == str.length() && j == pat.length()) return true;
12.        if (i == str.length() || j == pat.length()) return false;
13.
14.        // get current pattern character
15.        char c = pat.charAt(j);
16.
17.        // if the pattern character exists
18.        if (map.containsKey(c)) {
19.          String s = map.get(c);
20.
21.          // then check if we can use it to match str[i...i+s.length()]
22.          if (!str.startsWith(s, i)) {
23.            return false;
24.          }
25.
26.          // if it can match, great, continue to match the rest
27.          return isMatch(str, i + s.length(), pat, j + 1, map, set);
28.        }
29.
30.        // pattern character does not exist in the map
31.        for (int k = i; k < str.length(); k++) {
32.          String p = str.substring(i, k + 1);
33.
34.          if (set.contains(p)) {
35.            break;
36.          }
37.
38.          // create or update it
39.          map.put(c, p);
40.          set.add(p);
41.
42.          // continue to match the rest
```

```
43.          if (isMatch(str, k + 1, pat, j + 1, map, set)) {
44.            return true;
45.          }
46.        }
47.
48.      // we've tried our best but still no luck
49.      // backtracking
50.      set.remove(map.get(c));
51.      map.remove(c);
52.
53.      return false;
54.    }
55.  }
```

Word Search

- Word Search

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 =

```
1.  [
2.    ['A','B','C','E'],
3.    ['S','F','C','S'],
4.    ['A','D','E','E']
5.  ]
```

word = `"ABCCED"`, -> returns `true`,

word = `"SEE"`, -> returns `true`,

word = `"ABCB"`, -> returns `false`.

Solution:

```
1.  public class Solution {
2.      int[] dx = {-1, 1, 0, 0};
3.      int[] dy = {0, 0, -1, 1};
4.
```

```
5.    public boolean exist(char[][] board, String word) {
6.       int n = board.length;
7.       int m = board[0].length;
8.
9.       boolean[][] used = new boolean[n][m];
10.
11.      for (int i = 0; i < n; i++) {
12.        for (int j = 0; j < m; j++) {
13.          if (search(board, n, m, i, j, used, 0, word)) {
14.            return true;
15.          }
16.        }
17.      }
18.
19.      return false;
20.    }
21.
22.    boolean search(char[][] board, int n, int m, int i, int j,
       boolean[][] used, int k, String word) {
23.      if (k == word.length()) {
24.        return true;
25.      }
26.
27.      if (i < 0 || i >= n || j < 0 || j >= m) {
28.        return false;
29.      }
30.
31.      if (board[i][j] != word.charAt(k)) {
32.        return false;
33.      }
34.
35.      if (used[i][j]) {
36.        return false;
37.      }
38.
39.      used[i][j] = true;
40.
41.      boolean res = false;
```

```
42.
43.      for (int d = 0; d < 4; d++) {
44.          if (search(board, n, m, i + dx[d], j + dy[d], used, k + 1,
    word)) {
45.              res = true;
46.              break;
47.          }
48.      }
49.
50.      used[i][j] = false;
51.
52.      return res;
53.    }
54.  }
```

Binary Indexed Tree

- Binary Indexed Tree

Binary Indexed Tree

Binary Search

- Binary Search

Binary Search

Find Minimum in Rotated Sorted Array II

Find Minimum in Rotated Sorted Array II

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

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.

The array may contain duplicates.

Solution:

```java
public class Solution {
  public int findMin(int[] a) {
    return search(a, 0, a.length - 1);
  }

  int search(int[] a, int lo, int hi) {
    // a[] is sorted
    if (lo > hi) {
      return a[0];
    }

    int mid = lo + (hi - lo) / 2;

    if (mid > 0 && a[mid - 1] > a[mid]) {
```

```
15.         return a[mid];
16.       }
17.
18.       if (mid < a.length - 1 && a[mid] > a[mid + 1]) {
19.         return a[mid + 1];
20.       }
21.
22.       if (a[lo] == a[mid] && a[mid] == a[hi]) {
23.         return Math.min(search(a, lo, mid - 1), search(a, mid + 1,
    hi));
24.       }
25.
26.       if (a[lo] <= a[mid]) {
27.         return search(a, mid + 1, hi);
28.       } else {
29.         return search(a, lo, mid - 1);
30.       }
31.     }
32.   }
```

Find Minimum in Rotated Sorted Array

- Find Minimum in Rotated Sorted Array

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.

Solution:

```java
public class Solution {
  public int findMin(int[] a) {
    return search(a, 0, a.length - 1);
  }

  int search(int[] a, int lo, int hi) {
    // a[] is sorted
    if (lo > hi) {
      return a[0];
    }

    int mid = lo + (hi - lo) / 2;

    if (mid > 0 && a[mid - 1] > a[mid]) {
      return a[mid];
    }

    if (mid < a.length - 1 && a[mid] > a[mid + 1]) {
      return a[mid + 1];
```

```
20.        }
21.
22.      if (a[mid] < a[hi]) {
23.          return search(a, lo, mid - 1);
24.      } else {
25.          return search(a, mid + 1, hi);
26.      }
27.   }
28. }
```

Find Peak Element

- Find Peak Element

Find Peak Element

A peak element is an element that is greater than its neighbors.

Given an input array where `num[i] ≠ 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 `num[-1] = num[n] = -∞`.

For example, in array `[1, 2, 3, 1]`, 3 is a peak element and your function should return the index number 2.

Note:

Your solution should be in logarithmic complexity.

Solution:

```
1.  public class Solution {
2.    public int findPeakElement(int[] nums) {
3.      return search(nums, 0, nums.length - 1);
4.    }
5.
6.    int search(int[] a, int lo, int hi) {
7.      if (lo > hi) {
```

```
 8.        return -1;
 9.     }
10.
11.     int mid = lo + (hi - lo) / 2;
12.
13.     if ((mid == 0 || a[mid - 1] < a[mid]) && (mid == a.length - 1
    || a[mid] > a[mid + 1])) {
14.        return mid;
15.     }
16.
17.     if (a[mid] < a[mid + 1]) {
18.        return search(a, mid + 1, hi);
19.     } else {
20.        return search(a, lo, mid - 1);
21.     }
22.   }
23. }
```

First Bad Version

- First Bad Version

First Bad Version

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

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

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

Solution:

```
1. /* The isBadVersion API is defined in the parent class
   VersionControl.
2.     boolean isBadVersion(int version); */
3.
4. public class Solution extends VersionControl {
5.   public int firstBadVersion(int n) {
6.     return search(1, n);
7.   }
8.
```

```
 9.    int search(int lo, int hi) {
10.      if (lo > hi)
11.        return 0;
12.
13.      int mid = lo + (hi - lo) / 2;
14.
15.      boolean isBad = isBadVersion(mid);
16.
17.      if (isBad && (mid == 0 || !isBadVersion(mid - 1)))
18.        return mid;
19.
20.      if (isBad)
21.        return search(lo, mid - 1);
22.      else
23.        return search(mid + 1, hi);
24.    }
25.  }
```

H-Index II

- H-Index II

H-Index II

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

Hint:

Expected runtime complexity is in O(log n) and the input is sorted.

Solution:

```java
public class Solution {
  public int hIndex(int[] citations) {
    int n = citations.length;
    int lo = 0, hi = n - 1;

    while (lo <= hi) {
      int mid = lo + (hi - lo) / 2;

      if (citations[mid] >= n - mid) {
        hi = mid - 1;
      } else {
        lo = mid + 1;
      }
    }

    return n - 1 - hi;
  }
}
```

Search a 2D Matrix

- Search a 2D Matrix

Search a 2D Matrix

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.

For example,

Consider the following matrix:

```
1.  [
2.    [1,   3,  5,  7],
3.    [10, 11, 16, 20],
4.    [23, 30, 34, 50]
5.  ]
```

Given target = `3`, return `true`.

Solution:

```
1.  public class Solution {
2.    public boolean searchMatrix(int[][] matrix, int target) {
3.      int n = matrix.length;
4.      int m = matrix[0].length;
5.
```

```
 6.        int i = 0;
 7.        int j = m - 1;
 8.
 9.        while (i < n && j >= 0) {
10.          if (target == matrix[i][j]) {
11.            return true;
12.          }
13.
14.          if (target < matrix[i][j]) {
15.            j--;
16.          } else {
17.            i++;
18.          }
19.        }
20.
21.        return false;
22.      }
23.  }
```

Search for a Range

- Search for a Range

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]` .

Solution:

```
1.  public class Solution {
2.    public int[] searchRange(int[] nums, int target) {
3.      int lo = lower(nums, target, 0, nums.length - 1);
4.
5.      if (lo == -1) {
6.        return new int[] {-1, -1};
7.      }
8.
9.      int hi = upper(nums, target, 0, nums.length - 1);
10.
11.     return new int[] {lo, hi};
12.   }
13.
```

```
14.    int lower(int[] nums, int target, int lo, int hi) {
15.      if (lo > hi) {
16.        return -1;
17.      }
18.
19.      int mid = lo + (hi - lo) / 2;
20.
21.      if (nums[mid] == target && (mid == 0 || nums[mid - 1] <
    target)) {
22.        return mid;
23.      }
24.
25.      if (nums[mid] >= target) {
26.        return lower(nums, target, lo, mid - 1);
27.      } else {
28.        return lower(nums, target, mid + 1, hi);
29.      }
30.    }
31.
32.    int upper(int[] nums, int target, int lo, int hi) {
33.      if (lo > hi) {
34.        return -1;
35.      }
36.
37.      int mid = lo + (hi - lo) / 2;
38.
39.      if (nums[mid] == target && (mid == nums.length - 1 || target <
    nums[mid + 1])) {
40.        return mid;
41.      }
42.
43.      if (nums[mid] <= target) {
44.        return upper(nums, target, mid + 1, hi);
45.      } else {
46.        return upper(nums, target, lo, mid - 1);
47.      }
48.    }
49.  }
```

Search in Rotated Sorted Array II

- Search in Rotated Sorted Array II

Search in Rotated Sorted Array II

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

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

Write a function to determine if a given target is in the array.

Solution:

```java
public class Solution {
  public boolean search(int[] A, int target) {
    return bsearch(A, target, 0, A.length - 1);
  }

  private boolean bsearch(int[] A, int target, int lo, int hi) {
    if (lo > hi) {
      return false;
    }

    int mid = lo + (hi - lo) / 2;

    if (A[mid] == target) {
      return true;
    }

    if (A[lo] < A[mid]) {
      if (target >= A[lo] && target < A[mid]) {
```

```
19.          return bsearch(A, target, lo, mid - 1);
20.        } else {
21.          return bsearch(A, target, mid + 1, hi);
22.        }
23.      } else if (A[lo] > A[mid]) {
24.        if (target > A[mid] && target <= A[hi]) {
25.          return bsearch(A, target, mid + 1, hi);
26.        } else {
27.          return bsearch(A, target, lo, mid - 1);
28.        }
29.      } else {
30.        return bsearch(A, target, lo + 1, hi);
31.      }
32.    }
33. }
```

Search in Rotated Sorted Array

- Search in Rotated Sorted Array

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.

Solution:

```
1.  public class Solution {
2.    public int search(int[] nums, int target) {
3.      int n = nums.length;
4.
5.      return search(nums, 0, n - 1, target);
6.    }
7.
8.    int search(int[] nums, int lo, int hi, int target) {
9.      if (lo > hi)
10.       return -1;
11.
12.      int mid = lo + (hi - lo) / 2;
13.
14.      if (nums[mid] == target) {
15.        return mid;
16.      }
17.
18.      if (nums[lo] <= nums[mid]) {
```

```
19.          if (nums[lo] <= target && target < nums[mid]) {
20.            return search(nums, lo, mid - 1, target);
21.          } else {
22.            return search(nums, mid + 1, hi, target);
23.          }
24.        } else {
25.          if (nums[mid] < target && target <= nums[hi]) {
26.            return search(nums, mid + 1, hi, target);
27.          } else {
28.            return search(nums, lo, mid - 1, target);
29.          }
30.        }
31.      }
32.  }
```

Search Insert Position

- Search Insert Position

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

Solution:

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

  int search(int[] nums, int target, int lo, int hi) {
    if (lo > hi) {
      return lo;
    }

```

```
11.        int mid = lo + (hi - lo) / 2;
12.
13.        if (nums[mid] == target) {
14.            return mid;
15.        }
16.
17.        if (target < nums[mid]) {
18.            return search(nums, target, lo, mid - 1);
19.        } else {
20.            return search(nums, target, mid + 1, hi);
21.        }
22.    }
23. }
```

Smallest Rectangle Enclosing Black Pixels

Smallest Rectangle Enclosing Black Pixels

An image is represented by a binary matrix with `0` as a white pixel and `1` as a black pixel. The black pixels are connected, i.e., there is only one black region. Pixels are connected horizontally and vertically. Given the location `(x, y)` of one of the black pixels, return the area of the smallest (axis-aligned) rectangle that encloses all black pixels.

For example, given the following image:

```
1.  [
2.    "0010",
3.    "0110",
4.    "0100"
5.  ]
```

and `x = 0`, `y = 2`,
Return `6`.

Solution:

```
1.  public class Solution {
2.    public int minArea(char[][] image, int x, int y) {
3.      int n = image.length, m = image[0].length;
4.      int x1 = x, y1 = y, x2 = x, y2 = y;
5.
```

```
  6.        int[] dx = {0, 0, -1, 1};
  7.        int[] dy = {-1, 1, 0, 0};
  8.
  9.        Queue<int[]> queue = new LinkedList<>();
 10.        Set<Integer> visited = new HashSet<>();
 11.
 12.        queue.add(new int[]{x, y});
 13.        visited.add(x * m + y);
 14.
 15.        while (!queue.isEmpty()) {
 16.          int[] p = queue.poll();
 17.
 18.          x = p[0]; y = p[1];
 19.
 20.          if (x < x1) x1 = x;
 21.          if (x > x2) x2 = x;
 22.
 23.          if (y < y1) y1 = y;
 24.          if (y > y2) y2 = y;
 25.
 26.          for (int i = 0; i < 4; i++) {
 27.            int nx = x + dx[i], ny = y + dy[i];
 28.
 29.            if (nx < 0 || nx >= n || ny < 0 || ny >= m || image[nx][ny]
    != '1') {
 30.              continue;
 31.            }
 32.
 33.            if (!visited.contains(nx * m + ny)) {
 34.              queue.add(new int[]{nx, ny});
 35.              visited.add(nx * m + ny);
 36.            }
 37.          }
 38.        }
 39.
 40.        return (x2 - x1 + 1) * (y2 - y1 + 1);
 41.    }
 42. }
```

Binary Search Tree

- Binary Search Tree

Binary Search Tree

Contains Duplicate III

- Contains Duplicate III

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 nums[i] and nums[j] is at most t and the difference between i and j is at most k.

Solution:

```
1.  public class Solution {
2.    public boolean containsNearbyAlmostDuplicate(int[] nums, int k,
   int t) {
3.      TreeSet<Integer> tree = new TreeSet<>();
4.
5.      for (int i = 0; i < nums.length; i++) {
6.        Integer ceil = tree.ceiling(nums[i] - t);
7.        Integer floor = tree.floor(nums[i] + t);
8.
9.        if ((ceil != null && ceil <= nums[i]) || (floor != null &&
   floor >= nums[i])) {
10.          return true;
11.        }
12.
13.        tree.add(nums[i]);
14.
15.        if (i >= k) {
16.          tree.remove(nums[i - k]);
17.        }
18.      }
19.
```

```
20.        return false;
21.     }
22.  }
```

Data Stream as Disjoint Intervals

- Data Stream as Disjoint Intervals

Data Stream as Disjoint Intervals

Given a data stream input of non-negative integers a1, a2, …, an, …, summarize the numbers seen so far as a list of disjoint intervals.

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

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

Follow up:

What if there are lots of merges and the number of disjoint intervals are small compared to the data stream's size?

Bit Manipulation

- Bit Manipulation

Bit Manipulation

Bitwise AND of Numbers Range

- Bitwise AND of Numbers Range

Bitwise AND of Numbers Range

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive.

For example, given the range [5, 7], you should return 4.

Solution:

```java
public class Solution {
  public int rangeBitwiseAnd(int m, int n) {
    return rangeBitwiseAnd(m, n, 0x80000000);
  }

  int rangeBitwiseAnd(int m, int n, int shift) {
    // edge case
    if (m == 0) return 0;

    // find highest bit of m
    while ((shift & m) == 0) {
      if ((shift & n) != 0) return 0;
      shift >>>= 1;
    }

    return shift + rangeBitwiseAnd(m - shift, n - shift, shift);
  }
}
```

Generalized Abbreviation

- Generalized Abbreviation

Generalized Abbreviation

Write a function to generate the generalized abbreviations of a word.

Example:

Given word = `"word"` , return the following list (order does not matter):

```
1.  ["word", "1ord", "w1rd", "wo1d", "wor1", "2rd", "w2d", "wo2",
    "1o1d", "1or1", "w1r1", "1o2", "2r1", "3d", "w3", "4"]
```

Maximum Product of Word Lengths

- Maximum Product of Word Lengths

Maximum Product of Word Lengths

Given a string array `words` , find the maximum value of `length(word[i]) * length(word[j])` where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

Example 1:

Given `["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]`

Return `16`

The two words can be `"abcw", "xtfn"` .

Example 2:

Given `["a", "ab", "abc", "d", "cd", "bcd", "abcd"]`

Return `4`

The two words can be `"ab", "cd"` .

Example 3:

Given `["a", "aa", "aaa", "aaaa"]`

Return `0`

No such pair of words.

Number of 1 Bits

- Number of 1 Bits

Number of 1 Bits

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

For example, the 32-bit integer '11' has binary representation `00000000000000000000000000001011` , so the function should return 3.

Solution:

```
public class Solution {
  // you need to treat n as an unsigned value
  public int hammingWeight(int n) {
    int count = 0;

    for (int i = 0; i < 32; i++) {
      if ((n & 1) != 0)
        count++;

      n >>>= 1;
    }

    return count;
  }
}
```

Power of Four

- Power of Four

Power of Four

Given an integer (signed 32 bits), write a function to check whether it is a power of 4.

Example:

Given num = 16, return true. Given num = 5, return false.

Follow up: Could you solve it without loops/recursion?

Reverse Bits

- Reverse Bits

Reverse Bits

Reverse bits of a given 32 bits unsigned integer.

For example, given input 43261596 (represented in binary as **00000010100101000001111010011100**), return 964176192 (represented in binary as **00111001011110000010100101000000**).

Follow up:

If this function is called many times, how would you optimize it?

Related problem: Reverse Integer

Solution:

```
1.  public class Solution {
2.    // you need treat n as an unsigned value
3.    public int reverseBits(int n) {
4.      int m = 0;
5.
6.      for (int i = 0; i < 32; i++) {
7.        m |= ((n >>> i) & 1) << (31 - i);
8.      }
9.
10.     return m;
11.   }
12. }
```

Single Number II

- Single Number II

Single Number II

Given an array of integers, every element appears three times except for one. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Solution:

```java
public class Solution {
  public int singleNumber(int[] nums) {
    int ones = 0, twos = 0, threes = 0;

    for (int i = 0; i < nums.length; i++) {
      // twos holds the num that appears twice
      twos |= ones & nums[i];

      // ones holds the num that appears once
      ones ^= nums[i];

      // threes holds the num that appears three times
      threes = ones & twos;

      // if num[i] appears three times
      // doing this will clear ones and twos
      ones &= ~threes;
      twos &= ~threes;
```

```
19.        }
20.
21.      return ones;
22.    }
23.  }
```

Single Number III

Single Number III

Given an array of numbers nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once.

For example:

Given `nums = [1, 2, 1, 3, 2, 5]`, return `[3, 5]`.

Note:

1. The order of the result is not important. So in the above example, [5, 3] is also correct.
2. Your algorithm should run in linear runtime complexity. Could you implement it using only constant space complexity?

Solution:

```java
1.  public class Solution {
2.    public int[] singleNumber(int[] nums) {
3.      int xor = 0, a = 0, b = 0;
4.
5.      for (int i = 0; i < nums.length; i++)
6.        xor ^= nums[i];
7.
8.      // Get its last set bit
9.      xor &= -xor;
```

```
10.        //xor = xor & ~ (xor - 1);
11.
12.        for (int i = 0; i < nums.length; i++) {
13.          if ((xor & nums[i]) != 0)
14.            a ^= nums[i];
15.          else
16.            b ^= nums[i];
17.        }
18.
19.        return new int[]{a, b};
20.      }
21.  }
```

Subsets

Subsets

Given a set of distinct integers, nums, return all possible subsets.

Note: The solution set must not contain duplicate subsets.

For example,

If nums = `[1,2,3]` , a solution is:

```
1.  [
2.    [3],
3.    [1],
4.    [2],
5.    [1,2,3],
6.    [1,3],
7.    [2,3],
8.    [1,2],
9.    []
10. ]
```

Solution:

```java
1.  public class Solution {
2.    public List<List<Integer>> subsets(int[] nums) {
3.      List<List<Integer>> res = new ArrayList<List<Integer>>();
4.      res.add(new ArrayList<Integer>());
5.
6.      Arrays.sort(nums);
```

```
 7.
 8.      for (int i = 0; i < nums.length; i++) {
 9.        int size = res.size();
10.
11.        for (int j = 0; j < size; j++) {
12.          List<Integer> sol = new ArrayList<Integer>(res.get(j));
13.          sol.add(nums[i]);
14.          res.add(sol);
15.        }
16.      }
17.
18.      return res;
19.    }
20.  }
```

Sum of Two Integers

- Sum of Two Integers

Sum of Two Integers

Calculate the sum of two integers a and b, but you are **not allowed** to use the operator `+` and `-` .

Example:

Given a = 1 and b = 2, return 3.

Brainteaser

- Brainteaser

Brainteaser

Nim Game

-

Nim Game

You are playing the following Nim Game with your friend: There is a heap of stones on the table, each time one of you take turns to remove 1 to 3 stones. The one who removes the last stone will be the winner. You will take the first turn to remove the stones.

Both of you are very clever and have optimal strategies for the game. Write a function to determine whether you can win the game given the number of stones in the heap.

For example, if there are 4 stones in the heap, then you will never win the game: no matter 1, 2, or 3 stones you remove, the last stone will always be removed by your friend.

Hint:

If there are 5 stones in the heap, could you figure out a way to remove the stones such that you will always be the winner?

Solution:

```
1.  public class Solution {
```

```
2.    public boolean canWinNim(int n) {
3.      return (n % 4 != 0);
4.    }
5.  }
```

Breadth-first Search

- Breadth-first Search

Breadth-first Search

Shortest Distance from All Buildings

- Shortest Distance from All Buildings

Shortest Distance from All Buildings

You want to build a house on an empty land which reaches all buildings in the shortest amount of distance. You can only move up, down, left and right. You are given a 2D grid of values 0, 1 or 2, where:

- Each 0 marks an empty land which you can pass by freely.
- Each 1 marks a building which you cannot pass through.
- Each 2 marks an obstacle which you cannot pass through.

For example, given three buildings at `(0,0)`, `(0,4)`, `(2,2)`, and an obstacle at `(0,2)`:

```
1. 1 - 0 - 2 - 0 - 1
2. |   |   |   |   |
3. 0 - 0 - 0 - 0 - 0
4. |   |   |   |   |
5. 0 - 0 - 1 - 0 - 0
```

The point `(1,2)` is an ideal empty land to build a house, as the total travel distance of `3+3+1=7` is minimal. So return `7`.

Note:

There will be at least one building. If it is not possible to build such house according to the above rules, return -1.

Walls and Gates

- Walls and Gates

Walls and Gates

You are given a m x n 2D grid initialized with these three possible values.

1. `-1` - A wall or an obstacle.
2. `0` - A gate.
3. `INF` - Infinity means an empty room. We use the value `2^31 - 1 = 2147483647` to represent `INF` as you may assume that the distance to a gate is less than `2147483647`.

Fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, it should be filled with `INF`.

For example, given the 2D grid:

```
1.  INF  -1   0  INF
2.  INF INF INF  -1
3.  INF  -1 INF  -1
4.    0  -1 INF INF
```

After running your function, the 2D grid should be:

```
1.   3  -1   0   1
2.   2   2   1  -1
3.   1  -1   2  -1
4.   0  -1   3   4
```

Solution:

```
1.  public class Solution {
2.    int[] dx = {-1, 1, 0, 0};
3.    int[] dy = {0, 0, -1, 1};
4.
5.    public void wallsAndGates(int[][] rooms) {
6.      if (rooms.length == 0 || rooms[0].length == 0)
7.        return;
8.
9.      int n = rooms.length, m = rooms[0].length;
10.
11.      for (int i = 0; i < n; i++) {
12.        for (int j = 0; j < m; j++) {
13.          if (rooms[i][j] == 0) {
14.            bfs(rooms, n, m, i, j);
15.          }
16.        }
17.      }
18.    }
19.
20.    void bfs(int[][] rooms, int n, int m, int i, int j) {
21.      Queue<Integer[]> queue = new LinkedList<>();
22.      queue.add(new Integer[]{i, j});
23.      queue.add(null);
24.
25.      int level = 1;
26.
27.      while (!queue.isEmpty()) {
28.        Integer[] p = queue.poll();
29.
30.        if (p != null) {
31.          for (int k = 0; k < 4; k++) {
32.            int x = p[0] + dx[k];
33.            int y = p[1] + dy[k];
34.
35.            if (x >= 0 && x < n && y >= 0 && y < m && rooms[x][y] >
```

```
   level) {
36.              rooms[x][y] = level;
37.              queue.add(new Integer[]{x, y});
38.           }
39.         }
40.       } else if (!queue.isEmpty()) {
41.         level++;
42.         queue.add(null);
43.       }
44.     }
45.   }
46. }
```

Depth-first Search

- Depth-first Search

Depth-first Search

Nested List Weight Sum II

Nested List Weight Sum II

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list — whose elements may also be integers or other lists.

Different from the previous question where weight is increasing from root to leaf, now the weight is defined from bottom up. i.e., the leaf level integers have weight 1, and the root level integers have the largest weight.

Example 1:

Given the list `[[1,1],2,[1,1]]` , return `8` . (four 1's at depth 1, one 2 at depth 2)

Example 2:

Given the list `[1,[4,[6]]]` , return `17` . (one 1 at depth 3, one 4 at depth 2, and one 6 at depth 1; `1*3 + 4*2 + 6*1 = 17`)

Solution:

```
1.
2.
```

Nested List Weight Sum

- Nested List Weight Sum

Nested List Weight Sum

Given a nested list of integers, return the sum of all integers in the list weighted by their depth.

Each element is either an integer, or a list — whose elements may also be integers or other lists.

Example 1:

Given the list `[[1,1],2,[1,1]]` , return `10` . (four 1's at depth 2, one 2 at depth 1)

Example 2:

Given the list `[1,[4,[6]]]` , return `27` . (one 1 at depth 1, one 4 at depth 2, and one 6 at depth 3; `1 + 4*2 + 6*3 = 27`)

Solution:

```
1.
2.
```

Remove Invalid Parentheses

- Remove Invalid Parentheses

Remove Invalid Parentheses

Remove the minimum number of invalid parentheses in order to make the input string valid. Return all possible results.

Note: The input string may contain letters other than the parentheses `(` and `)` .

Examples:

```
1. "()())()" -> ["()()()", "(())()"]
2. "(a)())()" -> ["(a)()()", "(a())()"]
3. ")(" -> [""]
```

Solution:

```java
1.  public class Solution {
2.    public List<String> removeInvalidParentheses(String s) {
3.      List<String> res = new ArrayList<>();
4.
5.      // sanity check
6.      if (s == null) return res;
7.
8.      Set<String> visited = new HashSet<>();
9.      Queue<String> queue = new LinkedList<>();
10.
11.     // initialize
12.     queue.add(s);
13.     visited.add(s);
14.
```

```
15.        boolean found = false;
16.
17.        while (!queue.isEmpty()) {
18.          s = queue.poll();
19.
20.          if (isValid(s)) {
21.            // found an answer, add to the result
22.            res.add(s);
23.            found = true;
24.          }
25.
26.          if (found) continue;
27.
28.          // generate all possible states
29.          for (int i = 0; i < s.length(); i++) {
30.            // we only try to remove left or right paren
31.            if (s.charAt(i) != '(' && s.charAt(i) != ')') continue;
32.
33.            String t = s.substring(0, i) + s.substring(i + 1);
34.
35.            if (!visited.contains(t)) {
36.              // for each state, if it's not visited, add it to the
    queue
37.              queue.add(t);
38.              visited.add(t);
39.            }
40.          }
41.        }
42.
43.        return res;
44.      }
45.
46.    // helper function checks if string s contains valid parantheses
47.    boolean isValid(String s) {
48.        int count = 0;
49.
50.        for (int i = 0; i < s.length(); i++) {
51.            char c = s.charAt(i);
```

```
52.          if (c == '(') count++;
53.          if (c == ')' && count-- == 0) return false;
54.       }
55.
56.    return count == 0;
57.   }
58. }
```

Design

- Design

Design

Design Hit Counter

- Design Hit Counter

Design Hit Counter

Design a hit counter which counts the number of hits received in the past 5 minutes.

Each function accepts a timestamp parameter (in seconds granularity) and you may assume that calls are being made to the system in chronological order (ie, the timestamp is monotonically increasing). You may assume that the earliest timestamp starts at 1.

It is possible that several hits arrive roughly at the same time.

Example:

```
 1.  HitCounter counter = new HitCounter();
 2.
 3.  // hit at timestamp 1.
 4.  counter.hit(1);
 5.
 6.  // hit at timestamp 2.
 7.  counter.hit(2);
 8.
 9.  // hit at timestamp 3.
10.  counter.hit(3);
11.
12.  // get hits at timestamp 4, should return 3.
13.  counter.getHits(4);
14.
```

```
15.   // hit at timestamp 300.
16.   counter.hit(300);
17.
18.   // get hits at timestamp 300, should return 4.
19.   counter.getHits(300);
20.
21.   // get hits at timestamp 301, should return 3.
22.   counter.getHits(301);
```

Follow up:

What if the number of hits per second could be very large? Does your design scale?

Design Tic-Tac-Toe

- Design Tic-Tac-Toe

Design Tic-Tac-Toe

Design a Tic-tac-toe game that is played between two players on a n x n grid.

You may assume the following rules:

1. A move is guaranteed to be valid and is placed on an empty block.
2. Once a winning condition is reached, no more moves is allowed.
3. A player who succeeds in placing n of their marks in a horizontal, vertical, or diagonal row wins the game.

Example:

```
 1.  Given n = 3, assume that player 1 is "X" and player 2 is "O" in the
     board.
 2.
 3.  TicTacToe toe = new TicTacToe(3);
 4.
 5.  toe.move(0, 0, 1); -> Returns 0 (no one wins)
 6.  |X| | |
 7.  | | | |    // Player 1 makes a move at (0, 0).
 8.  | | | |
 9.
10.  toe.move(0, 2, 2); -> Returns 0 (no one wins)
11.  |X| |O|
12.  | | | |    // Player 2 makes a move at (0, 2).
```

```
13.  | | | |
14.
15.  toe.move(2, 2, 1); -> Returns 0 (no one wins)
16.  |X| |O|
17.  | | | |    // Player 1 makes a move at (2, 2).
18.  | | |X|
19.
20.  toe.move(1, 1, 2); -> Returns 0 (no one wins)
21.  |X| |O|
22.  | |O| |    // Player 2 makes a move at (1, 1).
23.  | | |X|
24.
25.  toe.move(2, 0, 1); -> Returns 0 (no one wins)
26.  |X| |O|
27.  | |O| |    // Player 1 makes a move at (2, 0).
28.  |X| |X|
29.
30.  toe.move(1, 0, 2); -> Returns 0 (no one wins)
31.  |X| |O|
32.  |O|O| |    // Player 2 makes a move at (1, 0).
33.  |X| |X|
34.
35.  toe.move(2, 1, 1); -> Returns 1 (player 1 wins)
36.  |X| |O|
37.  |O|O| |    // Player 1 makes a move at (2, 1).
38.  |X|X|X|
```

Follow up:

Could you do better than O(n^2) per `move()` operation?

Hint:

Could you trade extra space such that `move()` operation can be done in O(1)?
You need two arrays: int rows[n], int cols[n], plus

two variables: diagonal, anti_diagonal.

Flatten 2D Vector

Flatten 2D Vector

Implement an iterator to flatten a 2d vector.

For example,

Given 2d vector =

```
1.  [
2.    [1,2],
3.    [3],
4.    [4,5,6]
5.  ]
```

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: `[1,2,3,4,5,6]` .

Hint:

1. How many variables do you need to keep track?
2. Two variables is all you need. Try with x and y.
3. Beware of empty rows. It could be the first few rows.
4. To write correct code, think about the invariant to maintain. What is it?
5. The invariant is x and y must always point to a valid point in the 2d vector. Should you maintain your invariant ahead of time or right

when you need it?

6. Not sure? Think about how you would implement hasNext(). Which is more complex?

7. Common logic in two different places should be refactored into a common method.

Follow up:

As an added challenge, try to code it using only iterators in C++ or iterators in Java.

Solution:

```java
 1.  public class Vector2D {
 2.    private Iterator<List<Integer>> i;
 3.    private Iterator<Integer> j;
 4.
 5.    public Vector2D(List<List<Integer>> vec2d) {
 6.      i = vec2d.iterator();
 7.    }
 8.
 9.    public int next() {
10.      hasNext();
11.      return j.next();
12.    }
13.
14.    public boolean hasNext() {
15.      while ((j == null || !j.hasNext()) && i.hasNext())
16.        j = i.next().iterator();
17.      return j != null && j.hasNext();
18.    }
19.  }
20.
21.  /**
22.   * Your Vector2D object will be instantiated and called as such:
23.   * Vector2D i = new Vector2D(vec2d);
```

```
24.    * while (i.hasNext()) v[f()] = i.next();
25.    */
```

LRU Cache

- LRU Cache

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.

Solution:

```
 1.  public class LRUCache {
 2.
 3.    int capacity;
 4.    Map<Integer, ListNode> map;
 5.    ListNode head;
 6.    ListNode tail;
 7.
 8.    public LRUCache(int cap) {
 9.      capacity = cap;
10.      map = new HashMap<>();
11.      head = new ListNode(0, 0);
```

```
12.       tail = new ListNode(0, 0);
13.       head.next = tail;
14.       tail.prev = head;
15.     }
16.
17.     public int get(int key) {
18.       if (!map.containsKey(key)) {
19.         return -1;
20.       }
21.
22.       ListNode node = map.get(key);
23.       promote(node);
24.       return node.val;
25.     }
26.
27.     public void set(int key, int val) {
28.       ListNode node;
29.
30.       if (map.containsKey(key)) {
31.         node = map.get(key);
32.         promote(node);
33.         node.val = val;
34.         return;
35.       }
36.
37.       if (map.size() == capacity) {
38.         ListNode last = tail.prev;
39.         map.remove(last.key);
40.         remove(last);
41.       }
42.
43.       node = new ListNode(key, val);
44.       map.put(key, node);
45.       insert(node);
46.     }
47.
48.     void remove(ListNode node) {
49.       node.prev.next = node.next;
```

```
50.      node.next.prev = node.prev;
51.    }
52.
53.    void insert(ListNode node) {
54.      node.prev = head;
55.      node.next = head.next;
56.      head.next.prev = node;
57.      head.next = node;
58.    }
59.
60.    void promote(ListNode node) {
61.      remove(node);
62.      insert(node);
63.    }
64.
65.    class ListNode {
66.      int key;
67.      int val;
68.      ListNode prev = null;
69.      ListNode next = null;
70.
71.      public ListNode(int k, int v) {
72.        key = k;
73.        val = v;
74.      }
75.    }
76.
77. }
```

Peeking Iterator

- Peeking Iterator

Peeking Iterator

Given an Iterator class interface with methods:
`next()` and `hasNext()` , design and implement a
PeekingIterator that support the peek() operation —
it essentially peek() at the element that will be
returned by the next call to next().

Here is an example. Assume that the iterator is
initialized to the beginning of the list: `[1, 2, 3]` .

Call `next()` gets you 1, the first element in the
list.

Now you call `peek()` and it returns 2, the next
element. Calling `next()` after that still return 2.

You call `next()` the final time and it returns 3,
the last element. Calling `hasNext()` after that
should return false.

Hint:

1. Think of "looking ahead". You want to cache the
   next element.
2. Is one variable sufficient? Why or why not?
3. Test your design with call order of `peek()` before
   `next()` vs `next()` before `peek()` .

4. For a clean implementation, check out Google's
   guava library source code.

Follow up: How would you extend your design to be
generic and work with all types, not just integer?

Solution:

```
1.  // Java Iterator interface reference:
2.  //
    https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html
3.  class PeekingIterator implements Iterator<Integer> {
4.      Integer num;
5.      Iterator<Integer> iterator;
6.
7.      public PeekingIterator(Iterator<Integer> iterator) {
8.        // initialize any member here.
9.        this.num = null;
10.       this.iterator = iterator;
11.     }
12.
13.     // Returns the next element in the iteration without advancing
    the iterator.
14.     public Integer peek() {
15.       if (num == null && iterator.hasNext()) {
16.         num = iterator.next();
17.       }
18.       return num;
19.     }
20.
21.     // hasNext() and next() should behave the same as in the Iterator
    interface.
22.     // Override them if needed.
23.     @Override
24.     public Integer next() {
25.       if (num == null) {
26.         return iterator.next();
```

```
27.        }
28.
29.      Integer tmp = num;
30.      num = null;
31.      return tmp;
32.    }
33.
34.    @Override
35.    public boolean hasNext() {
36.      return num != null || iterator.hasNext();
37.    }
38.  }
```

Zigzag Iterator

-

Zigzag Iterator

Given two 1d vectors, implement an iterator to return their elements alternately.

For example, given two 1d vectors:

```
1.  v1 = [1, 2]
2.  v2 = [3, 4, 5, 6]
```

By calling next repeatedly until hasNext returns `false` , the order of elements returned by next should be: `[1, 3, 2, 4, 5, 6]` .

Follow up: What if you are given k 1d vectors? How well can your code be extended to such cases?

The "Zigzag" order is not clearly defined and is ambiguous for `k > 2` cases. If "Zigzag" does not look right to you, replace "Zigzag" with "Cyclic". For example, given the following input:

```
1.  [1,2,3]
2.  [4,5,6,7]
3.  [8,9]
```

It should return `[1,4,8,2,5,9,3,6,7]` .

Solution:

```java
public class ZigzagIterator {
    int i1 = 0;
    int i2 = 0;

    boolean flag = false;

    List<Integer> l1;
    List<Integer> l2;

    public ZigzagIterator(List<Integer> v1, List<Integer> v2) {
        l1 = v1;
        l2 = v2;
    }

    public int next() {
        flag = !flag;

        if (i1 < l1.size() && (flag  || i2 >= l2.size()))
            return l1.get(i1++);

        if (i2 < l2.size() && (!flag || i1 >= l1.size()))
            return l2.get(i2++);

        return -1;
    }

    public boolean hasNext() {
        return i1 < l1.size() || i2 < l2.size();
    }
}

/**
 * Your ZigzagIterator object will be instantiated and called as such:
 * ZigzagIterator i = new ZigzagIterator(v1, v2);
 * while (i.hasNext()) v[f()] = i.next();
 */
```

Divide and Conquer

- Divide and Conquer

Divide and Conquer

Count of Range Sum

- Count of Range Sum

Count of Range Sum

Given an integer array `nums`, return the number of range sums that lie in `[lower, upper]` inclusive. Range sum `S(i, j)` is defined as the sum of the elements in `nums` between indices `i` and `j` (`i` ≤ `j`), inclusive.

Note:
A naive algorithm of O(n^2) is trivial. You MUST do better than that.

Example:

Given nums = `[-2, 5, -1]`, lower = `-2`, upper = `2`,
Return `3`.
The three ranges are : `[0, 0], [2, 2], [0, 2]` and their respective sums are: `-2, -1, 2`.

Different Ways to Add Parentheses

Different Ways to Add Parentheses

Given a string of numbers and operators, return all possible results from computing all the different possible ways to group numbers and operators. The valid operators are `+`, `-` and `*`.

Example 1

Input: `"2-1-1"` .

```
1. ((2-1)-1) = 0
2. (2-(1-1)) = 2
```

Output: `[0, 2]`

Example 2

Input: `"2*3-4*5"`

```
1. (2*(3-(4*5))) = -34
2. ((2*3)-(4*5)) = -14
3. ((2*(3-4))*5) = -10
4. (2*((3-4)*5)) = -10
5. (((2*3)-4)*5) = 10
```

Output: `[-34, -14, -10, -10, 10]`

Solution:

```
public class Solution {
  public List<Integer> diffWaysToCompute(String input) {
    List<Integer> res = new ArrayList<Integer>();

    for (int i = 1; i < input.length(); i++) {
      char c = input.charAt(i);

      if (c == '+' || c == '-' || c == '*') {
        String s1 = input.substring(0, i);
        String s2 = input.substring(i + 1);

        List<Integer> l1 = diffWaysToCompute(s1);
        List<Integer> l2 = diffWaysToCompute(s2);

        for (Integer n1 : l1) {
          for (Integer n2 : l2) {
            switch (c) {
              case '+': res.add(n1 + n2); break;
              case '-': res.add(n1 - n2); break;
              case '*': res.add(n1 * n2); break;
            }
          }
        }
      }
    }

    if (res.size() == 0) {
      res.add(Integer.valueOf(input));
    }

    return res;
  }
}
```

Expression Add Operators

- Expression Add Operators

Expression Add Operators

Given a string that contains only digits `0-9` and a target value, return all possibilities to add binary operators (not unary) `+`, `-`, or `*` between the digits so they evaluate to the target value.

Examples:

```
1.  "123", 6 -> ["1+2+3", "1*2*3"]
2.  "232", 8 -> ["2*3+2", "2+3*2"]
3.  "105", 5 -> ["1*0+5","10-5"]
4.  "00", 0 -> ["0+0", "0-0", "0*0"]
5.  "3456237490", 9191 -> []
```

Solution:

```java
public class Solution {
  public List<String> addOperators(String num, int target) {
    List<String> res = new ArrayList<>();

    if (num == null || num.length() == 0)
      return res;

    helper(num, target, 0, 0, 0, "", res);

    return res;
  }

  public void helper(String num, int target, int index, long total,
    long last, String sol, List<String> res){
```

```
14.        if (index == num.length()) {
15.          if (target == total)
16.            res.add(sol);
17.
18.          return;
19.        }
20.
21.        for (int i = index; i < num.length(); i++) {
22.          // for example, input is "105", we don't need answer like
     "1*05"
23.          if (i > index && num.charAt(index) == '0')
24.            break;
25.
26.          long curr = Long.parseLong(num.substring(index, i + 1));
27.
28.          if (index == 0) {
29.            // 第一个数
30.            helper(num, target, i + 1, curr, curr, sol + curr, res);
31.          } else {
32.            // 不是第一个数，我们可以添加运算符了
33.            helper(num, target, i + 1, total + curr,  curr, sol + "+" +
     curr, res);
34.            helper(num, target, i + 1, total - curr, -curr, sol + "-" +
     curr, res);
35.            helper(num, target, i + 1, total - last + last * curr, last
     * curr, sol + "*" + curr, res);
36.          }
37.        }
38.      }
39.  }
```

Majority Element

- Majority Element

Majority Element

Given an array of size n, find the majority element. The majority element is the element that appears more than ⌊ n/2 ⌋ times.

You may assume that the array is non-empty and the majority element always exist in the array.

Solution:

```java
public class Solution {
  public int majorityElement(int[] nums) {
    Integer m = null;
    int c = 0;

    for (int i = 0; i < nums.length; i++) {
      if (m != null && m == nums[i]) {
        c++;
      } else if (c == 0) {
        m = nums[i];
        c = 1;
      } else {
        c--;
      }
    }

    c = 0;
    for (int i = 0; i < nums.length; i++) {
      if (m != null && m == nums[i]) {
        c++;
```

```
21.          }
22.      }
23.
24.      if (c > nums.length / 2) {
25.          return m;
26.      } else {
27.          return -1;
28.      }
29.   }
30. }
```

Median of Two Sorted Arrays

- Median of Two Sorted Arrays

Median of Two Sorted Arrays

There are two sorted arrays nums1 and nums2 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)).

Example 1:

```
1.  nums1 = [1, 3]
2.  nums2 = [2]
3.
4.  The median is 2.0
```

Example 2:

```
1.  nums1 = [1, 2]
2.  nums2 = [3, 4]
3.
4.  The median is (2 + 3)/2 = 2.5
```

Solution:

```
1.  public class Solution {
2.    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
3.      int l1 = nums1.length;
4.      int l2 = nums2.length;
5.
6.      int k = (l1 + l2) / 2;
7.
```

```
8.        if ((l1 + l2) % 2 == 0) {
9.            return (helper(nums1, 0, l1 - 1, nums2, 0, l2 - 1, k) +
    helper(nums1, 0, l1 - 1, nums2, 0, l2 - 1, k + 1)) / 2.0;
10.        } else {
11.            return helper(nums1, 0, l1 - 1, nums2, 0, l2 - 1, k + 1);
12.        }
13.    }
14.
15.    double helper(int[] a1, int i1, int j1, int[] a2, int i2, int j2,
    int k) {
16.        int l1 = j1 - i1 + 1;
17.        int l2 = j2 - i2 + 1;
18.
19.        if (l1 > l2) {
20.            return helper(a2, i2, j2, a1, i1, j1, k);
21.        }
22.
23.        if (l1 == 0) {
24.            return a2[i2 + k - 1];
25.        }
26.
27.        if (k == 1) {
28.            return Math.min(a1[i1], a2[i2]);
29.        }
30.
31.        int n1 = Math.min(k / 2, l1);
32.        int n2 = k - n1;
33.
34.        int v1 = a1[i1 + n1 - 1];
35.        int v2 = a2[i2 + n2 - 1];
36.
37.        if (v1 == v2) {
38.            return v1;
39.        } else if (v1 < v2) {
40.            return helper(a1, i1 + n1, j1, a2, i2, i2 + n2 - 1, k - n1);
41.        } else {
42.            return helper(a1, i1, i1 + n1 - 1, a2, i2 + n2, j2, k - n2);
43.        }
```

```
44.    }
45.  }
```

Search a 2D Matrix II

-

Search a 2D Matrix II

Write an efficient algorithm that searches for a value in an m x 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.  [
2.    [1,   4,  7, 11, 15],
3.    [2,   5,  8, 12, 19],
4.    [3,   6,  9, 16, 22],
5.    [10, 13, 14, 17, 24],
6.    [18, 21, 23, 26, 30]
7.  ]
```

Given **target** = `5`, return `true`.

Given **target** = `20`, return `false`.

Solution:

```
1.  public class Solution {
```

```
 2.    public boolean searchMatrix(int[][] matrix, int target) {
 3.      int n = matrix.length, m = matrix[0].length;
 4.
 5.      int i = 0, j = m - 1;
 6.
 7.      while (i < n && j >= 0) {
 8.        if (matrix[i][j] == target)
 9.          return true;
10.
11.        if (target < matrix[i][j])
12.          j--;
13.        else
14.          i++;
15.      }
16.
17.      return false;
18.    }
19.  }
```

Dynamic Programming

- Dynamic Programming

Dynamic Programming

Android Unlock Patterns

-

Android Unlock Patterns

Given an Android **3x3** key lock screen and two integers **m** and **n**, where 1 ≤ m ≤ n ≤ 9, count the total number of unlock patterns of the Android lock screen, which consist of minimum of m keys and maximum n keys.

Rules for a valid pattern:

1. Each pattern must connect at least m keys and at most n keys.
2. All the keys must be distinct.
3. If the line connecting two consecutive keys in the pattern passes through any other keys, the other keys must have previously selected in the pattern. No jumps through non selected key is allowed.
4. The order of keys used matters.
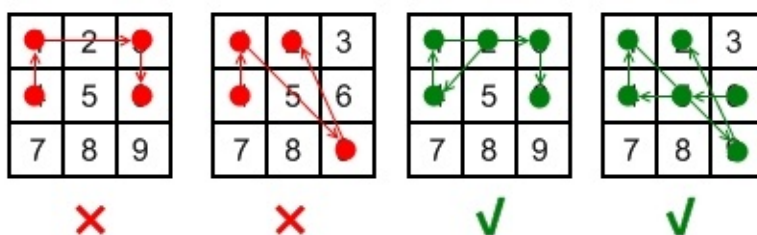
Explanation:

```
1.  | 1 | 2 | 3 |
2.  | 4 | 5 | 6 |
3.  | 7 | 8 | 9 |
```

Invalid move: 4 - 1 - 3 - 6

Line 1 - 3 passes through key 2 which had not been selected in the pattern.

Invalid move: 4 - 1 - 9 - 2

Line 1 - 9 passes through key 5 which had not been selected in the pattern.

Valid move: 2 - 4 - 1 - 3 - 6

Line 1 - 3 is valid because it passes through key 2, which had been selected in the pattern

Valid move: 6 - 5 - 4 - 1 - 9 - 2

Line 1 - 9 is valid because it passes through key 5, which had been selected in the pattern.

Example:

Given m = 1, n = 1, return 9.

Best Time to Buy and Sell Stock III

- Best Time to Buy and Sell Stock III

Best Time to Buy and Sell Stock III

Say you have an array for which the i-th element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most two transactions.

Note:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution:

```
1.  public class Solution {
2.    public int maxProfit(int[] prices) {
3.      if (prices == null || prices.length == 0) {
4.        return 0;
5.      }
6.
7.      int n = prices.length;
8.      int profit = 0;
9.
10.     // scan from left
11.     // left[i] keeps the max profit from 0 to i
12.     int[] left = new int[n];
13.     int min = prices[0];
14.
15.     for (int i = 1; i < n; i++) {
```

```
16.          left[i] = Math.max(left[i - 1], prices[i] - min);
17.          min = Math.min(min, prices[i]);
18.      }
19.
20.      // scan from right
21.      // right[i] keeps the max profit from i to n - 1
22.      int[] right = new int[n];
23.      int max = prices[n - 1];
24.
25.      for (int i = n - 2; i >= 0; i--) {
26.          right[i] = Math.max(right[i + 1], max - prices[i]);
27.          max = Math.max(max, prices[i]);
28.
29.          profit = Math.max(profit, left[i] + right[i]);
30.      }
31.
32.      return profit;
33.    }
34.  }
```

Time complexity: O(n)

Best Time to Buy and Sell Stock IV

- Best Time to Buy and Sell Stock IV

Best Time to Buy and Sell Stock IV

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most k transactions.

Note:

You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution:

```
1.  public class Solution {
2.    public int maxProfit(int k, int[] prices) {
3.      int n = prices.length;
4.
5.      if (k >= n) {
6.          return solveMaxProfit(prices);
7.      }
8.
9.      // The local array tracks maximum profit of j transactions &
    the last transaction is on ith day.
10.      // The global array tracks the maximum profit of j transactions
    until ith day.
11.      int[][] local = new int[n][k + 1];
12.      int[][] global = new int[n][k + 1];
13.
```

```
14.      for (int i = 1; i < n; i++) {
15.        int diff = prices[i] - prices[i - 1];
16.
17.        for (int j = 1; j <= k; j++) {
18.          local[i][j] = Math.max(
19.            global[i - 1][j - 1] + Math.max(diff, 0),
20.            local[i - 1][j] + diff
21.          );
22.
23.          global[i][j] = Math.max(global[i - 1][j], local[i][j]);
24.        }
25.      }
26.
27.      return global[n - 1][k];
28.    }
29.
30.    private int solveMaxProfit(int[] prices) {
31.      int profit = 0;
32.
33.      for (int i = 1; i < prices.length; i++) {
34.        // as long as there is a price gap, we gain a profit.
35.        if (prices[i] > prices[i - 1]) {
36.          profit += prices[i] - prices[i - 1];
37.        }
38.      }
39.
40.      return profit;
41.    }
42.  }
```

Best Time to Buy and Sell Stock with Cooldown

- Best Time to Buy and Sell Stock with Cooldown

Best Time to Buy and Sell Stock with Cooldown

Say you have an array for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

- You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
- After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Example:

```
1.  prices = [1, 2, 3, 0, 2]
2.  maxProfit = 3
3.  transactions = [buy, sell, cooldown, buy, sell]
```

Solution:

```
1.
2.
```

Best Time to Buy and Sell Stock

- Best Time to Buy and Sell Stock

Best Time to Buy and Sell Stock

Say you have an array for which the i-th element is the price of a given stock on day i.

If you were only permitted to complete at most one transaction (ie, buy one and sell one share of the stock), design an algorithm to find the maximum profit.

Example 1:

```
1.  Input: [7, 1, 5, 3, 6, 4]
2.  Output: 5
3.
4.  max. difference = 6-1 = 5 (not 7-1 = 6, as selling price needs to
    be larger than buying price)
```

Example 2:

```
1.  Input: [7, 6, 4, 3, 1]
2.  Output: 0
3.
4.  In this case, no transaction is done, i.e. max profit = 0.
```

Solution:

```
1.  public class Solution {
2.    public int maxProfit(int[] prices) {
3.      int min = Integer.MAX_VALUE;
```

```
 4.        int max = 0;
 5.
 6.        for (int i = 0; i < prices.length; i++) {
 7.          max = Math.max(max, prices[i] - min);
 8.          min = Math.min(min, prices[i]);
 9.        }
10.
11.        return max;
12.    }
13. }
```

Time complexity: O(n)

Bomb Enemy

- Bomb Enemy

Bomb Enemy

Given a 2D grid, each cell is either a wall `'W'`, an enemy `'E'` or empty `'0'` (the number zero), return the maximum enemies you can kill using one bomb.

The bomb kills all the enemies in the same row and column from the planted point until it hits the wall since the wall is too strong to be destroyed.

Note that you can only put the bomb at an empty cell.

Example:

For the given grid

```
1.  0 E 0 0
2.  E 0 W E
3.  0 E 0 0
4.
5.  return 3. (Placing a bomb at (1,1) kills 3 enemies)
```

Solution:

```
1.
2.
```

Burst Balloons

- Burst Balloons

Burst Balloons

Given n balloons, indexed from 0 to n-1. Each balloon is painted with a number on it represented by array nums. You are asked to burst all the balloons. If the you burst balloon i you will get `nums[left] * nums[i] * nums[right]` coins. Here `left` and `right` are adjacent indices of i. After the burst, the `left` and `right` then becomes adjacent.

Find the maximum coins you can collect by bursting the balloons wisely.

Note:

1. You may imagine `nums[-1] = nums[n] = 1`. They are not real therefore you can not burst them.
2. $0 \leq$ `n` $\leq 500, 0 \leq$ `nums[i]` ≤ 100

Example:

Given `[3, 1, 5, 8]`

Return `167`

```
1.     nums = [3,1,5,8] --> [3,5,8] -->  [3,8]   --> [8]  --> []
2.    coins =  3*1*5     +  3*5*8    +  1*3*8     + 1*8*1   = 167
```

Solution:

```
1.
2.
```

Climbing Stairs

- Climbing Stairs

Climbing Stairs

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Solution:

```java
public class Solution {
  public int climbStairs(int n) {
    if (n == 0)
      return 0;

    if (n == 1)
      return 1;

    int[] dp = new int[n + 1];

    dp[0] = 1; dp[1] = 1;

    for (int i = 2; i <= n; i++) {
      dp[i] = dp[i - 1] + dp[i - 2];
    }

    return dp[n];
  }
}
```

Coin Change

- Coin Change

Coin Change

You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1` .

Example 1:

coins = `[1, 2, 5]` , amount = `11`
return `3` (11 = 5 + 5 + 1)

Example 2:

coins = `[2]` , amount = `3`
return `-1` .

Note:

You may assume that you have an infinite number of each kind of coin.

Counting Bits

- Counting Bits

Counting Bits

Given a non negative integer number num. For every numbers i in the range **0 ≤ i ≤ num** calculate the number of 1's in their binary representation and return them as an array.

Example:

For `num = 5` you should return `[0,1,1,2,1,2]` .

Follow up:

- It is very easy to come up with a solution with run time **O(n*sizeof(integer))**. But can you do it in linear time **O(n)** /possibly in a single pass?
- Space complexity should be **O(n)**.
- Can you do it like a boss? Do it without using any builtin function like __**builtin_popcount** in c++ or in any other language.

Hint:

1. You should make use of what you have produced already.
2. Divide the numbers in ranges like `[2-3]` , `[4-7]` , `[8-15]` and so on. And try to generate new range from previous.

3. Or does the odd/even status of the number help
    you in calculating the number of 1s?

Dungeon Game

- Dungeon Game

Dungeon Game

The demons had captured the princess (P) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of M x N rooms laid out in a 2D grid. Our valiant knight (K) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.

The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.

Some of the rooms are guarded by demons, so the knight loses health (negative integers) upon entering these rooms; other rooms are either empty (0's) or contain magic orbs that increase the knight's health (positive integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.

For example, given the dungeon below, the initial health of the knight must be at least 7 if he follows the optimal path `RIGHT-> RIGHT -> DOWN -> DOWN` .

-2 (K)	-3	3
-5	-10	1
10	30	-5 (P)

Notes:

- The knight's health has no upper bound.
- Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

Solution:

```java
public class Solution {
  public int calculateMinimumHP(int[][] dungeon) {
    int n = dungeon.length, m = dungeon[0].length;

    // dp(i, j) represents the minimum blood needed at cell(i, j)
    int[][] dp = new int[n][m];

    // initialize the last cell
    dp[n - 1][m - 1] = Math.max(1, 1 - dungeon[n - 1][m - 1]);

    // last row
    for (int j = m - 2; j >= 0; j--) {
      dp[n - 1][j] = Math.max(1, dp[n - 1][j + 1] - dungeon[n - 1]
```

```
       [j]);
14.      }
15.

16.      // last col
17.      for (int i = n - 2; i >= 0; i--) {
18.        dp[i][m - 1] = Math.max(1, dp[i + 1][m - 1] - dungeon[i][m -
     1]);
19.      }
20.

21.      // other cells
22.      for (int i = n - 2; i >= 0; i--) {
23.        for (int j = m - 2; j >= 0; j--) {
24.          dp[i][j] = Math.max(1, Math.min(dp[i + 1][j], dp[i][j + 1])
     - dungeon[i][j]);
25.        }
26.      }
27.

28.      return dp[0][0];
29.    }
30.  }
```

House Robber II

- House Robber II

House Robber II

Note: This is an extension of House Robber.

After robbing those houses on that street, the thief has found himself a new place for his thievery so that he will not get too much attention. This time, all houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, the security system for these houses remain the same as for those in the previous street.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

Solution:

```java
public class Solution {
    public int rob(int[] nums) {
        if (nums == null || nums.length == 0)
            return 0;

        int n = nums.length;

        if (n == 1)
            return nums[0];
```

```
10.
11.     if (n == 2)
12.       return Math.max(nums[0], nums[1]);
13.
14.     return Math.max(rob(nums, 0, n - 2), rob(nums, 1, n - 1));
15.   }
16.
17.   int rob(int[] nums, int s, int e) {
18.     int n = e - s + 1;
19.
20.     int[] dp = new int[n + 1];
21.     dp[1] = nums[s];
22.
23.     for (int i = 2; i <= n; i++) {
24.       dp[i] = Math.max(dp[i - 1], nums[s - 1 + i] + dp[i - 2]);
25.     }
26.
27.     return dp[n];
28.   }
29. }
```

House Robber

- House Robber

House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security system connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can rob tonight without alerting the police.

Solution:

```java
public class Solution {
  public int rob(int[] a) {
    if (a == null || a.length == 0)
      return 0;

    int n = a.length;

    // dp(i) represents the max money by robbing till i-th house
    int[] dp = new int[n + 1];

    dp[1] = a[0];

```

```
13.        for (int i = 2; i <= n; i++) {
14.           dp[i] = Math.max(dp[i - 1], a[i - 1] + dp[i - 2]);
15.        }
16.
17.        return dp[n];
18.     }
19.  }
```

Longest Increasing Subsequence

Longest Increasing Subsequence

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

For example,

Given `[10, 9, 2, 5, 3, 7, 101, 18]` ,
The longest increasing subsequence is `[2, 3, 7, 101]` , therefore the length is `4` . Note that there may be more than one LIS combination, it is only necessary for you to return the length.

Your algorithm should run in O(n2) complexity.

Follow up: Could you improve it to O(n log n) time complexity?

Solution:

```
1.   public class Solution {
2.     public int lengthOfLIS(int[] nums) {
3.       if (nums.length == 0) return 0;
4.
5.       int n = nums.length, max = 0;
6.       int[] dp = new int[n];
7.       Arrays.fill(dp, 1);
8.
9.       for (int i = 0; i < n; i++) {
10.        for (int j = 0; j < i; j++)
```

```
11.            if (nums[i] > nums[j] && dp[j] + 1 > dp[i])
12.              dp[i] = dp[j] + 1;
13.
14.          max = Math.max(max, dp[i]);
15.        }
16.
17.        return max;
18.      }
19.  }
```

Maximal Square

- Maximal Square

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:

```
1. 1 0 1 0 0
2. 1 0 1 1 1
3. 1 1 1 1 1
4. 1 0 0 1 0
```

Return 4.

Solution:

```
1.  public class Solution {
2.    public int maximalSquare(char[][] a) {
3.        if (a == null || a.length == 0 || a[0].length == 0)
4.          return 0;
5.
6.        int max = 0;
7.        int n = a.length;
8.        int m = a[0].length;
9.
10.       // recurrence formula:
11.       // dp(i, j) = min{ dp(i-1, j-1), dp(i-1, j), dp(i, j-1) }
12.       int[][] dp = new int[n + 1][m + 1];
13.
14.       for (int i = 1; i <= n; i++) {
```

```
15.            for (int j = 1; j <= m; j++) {
16.              if (a[i - 1][j - 1] == '1') {
17.                dp[i][j] = Math.min(
18.                  dp[i - 1][j - 1],
19.                  Math.min(dp[i - 1][j], dp[i][j - 1])
20.                ) + 1;
21.                max = Math.max(max, dp[i][j]);
22.              }
23.            }
24.          }
25.
26.          // return the area
27.          return max * max;
28.      }
29.  }
```

Maximum Product Subarray

- Maximum Product Subarray

Maximum Product Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest product.

For example, given the array `[2,3,-2,4]`, the contiguous subarray `[2,3]` has the largest product = `6`.

Solution:

```
public class Solution {
  public int maxProduct(int[] nums) {
    int n = nums.length;

    int res = nums[0];
    int min = nums[0];
    int max = nums[0];

    for (int i = 1; i < n; i++) {
      if (nums[i] > 0) {
        max = Math.max(nums[i], max * nums[i]);
        min = Math.min(nums[i], min * nums[i]);
      } else {
        int tmp = max;
        max = Math.max(nums[i], min * nums[i]);
        min = Math.min(nums[i], tmp * nums[i]);
      }

      res = Math.max(res, max);
```

```
20.        }
21.
22.        return res;
23.    }
24. }
```

Maximum Subarray

- Maximum Subarray

Maximum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array `[-2,1,-3,4,-1,2,1,-5,4]`, the contiguous subarray `[4,-1,2,1]` has the largest sum = `6`.

More practice:

If you have figured out the *O(n)* solution, try coding another solution using the divide and conquer approach, which is more subtle.

Solution:

```
1.  public class Solution {
2.    public int maxSubArray(int[] nums) {
3.      return maxSubArray(nums, 0, nums.length - 1);
4.    }
5.
6.    int maxSubArray(int[] nums, int lo, int hi) {
7.      if (lo == hi) {
8.        return nums[lo];
9.      }
10.
11.     int mid = lo + (hi - lo) / 2;
12.
```

```
13.        int left = maxSubArray(nums, lo, mid);
14.        int right = maxSubArray(nums, mid + 1, hi);
15.        int middle = maxMiddleSum(nums, lo, mid, hi);
16.
17.        return Math.max(middle, Math.max(left, right));
18.    }
19.
20.    int maxMiddleSum(int[] nums, int lo, int mid, int hi) {
21.        int sum = 0;
22.
23.        int left = Integer.MIN_VALUE;
24.        for (int i = mid; i >= lo; i--) {
25.            sum += nums[i];
26.            left = Math.max(left, sum);
27.        }
28.
29.        sum = 0;
30.
31.        int right = Integer.MIN_VALUE;
32.        for (int i = mid + 1; i <= hi; i++) {
33.            sum += nums[i];
34.            right = Math.max(right, sum);
35.        }
36.
37.        return left + right;
38.    }
39. }
```

Minimum Path Sum

- Minimum Path Sum

Minimum Path Sum

Given a m x 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.

Note: You can only move either down or right at any point in time.

Solution:

```
public class Solution {
  public int minPathSum(int[][] grid) {
    int n = grid.length, m = grid[0].length;

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

    dp[0][0] = grid[0][0];

    // first row
    for (int j = 1; j < m; j++)
      dp[0][j] = grid[0][j] + dp[0][j - 1];

    // first col
    for (int i = 1; i < n; i++)
      dp[i][0] = grid[i][0] + dp[i - 1][0];

    for (int i = 1; i < n; i++) {
      for (int j = 1; j < m; j++) {
        dp[i][j] = grid[i][j] + Math.min(dp[i - 1][j], dp[i][j -
1]);
```

```
20.          }
21.       }
22.
23.       return dp[n - 1][m - 1];
24.    }
25. }
```

Paint Fence

- Paint Fence

Paint Fence

There is a fence with n posts, each post can be painted with one of the k colors.

You have to paint all the posts such that no more than two adjacent fence posts have the same color.

Return the total number of ways you can paint the fence.

Note:

n and k are non-negative integers.

Solution:

```
 1.  public class Solution {
 2.    public int numWays(int n, int k) {
 3.      if (n == 0)
 4.        return 0;
 5.
 6.      if (n == 1)
 7.        return k;
 8.
 9.      int same = k;
10.      int diff = k * (k - 1);
11.
12.      for (int i = 2; i < n; i++) {
13.        int tmp = diff;
14.        diff = (diff + same) * (k - 1);
```

```
15.          same = tmp;
16.      }
17.
18.      return diff + same;
19.    }
20. }
```

Paint House II

- Paint House II

Paint House II

There are a row of n houses, each house can be painted with one of the k colors. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a `n x k` cost matrix. For example, `costs[0][0]` is the cost of painting house 0 with color 0; `costs[1][2]` is the cost of painting house 1 with color 2, and so on… Find the minimum cost to paint all houses.

Note:

All costs are positive integers.

Follow up:

Could you solve it in O(nk) runtime?

Solution:

```
1.  public class Solution {
2.    public int minCostII(int[][] costs) {
3.      if (costs == null || costs.length == 0) return 0;
4.
```

```
5.        int n = costs.length, k = costs[0].length;
6.        // min1 is the index of the 1st-smallest cost till previous
    house
7.        // min2 is the index of the 2nd-smallest cost till previous
    house
8.        int min1 = -1, min2 = -1;
9.
10.       for (int i = 0; i < n; i++) {
11.         int last1 = min1, last2 = min2;
12.         min1 = -1; min2 = -1;
13.
14.         for (int j = 0; j < k; j++) {
15.           if (j != last1) {
16.             costs[i][j] += last1 < 0 ? 0 : costs[i - 1][last1];
17.           } else {
18.             costs[i][j] += last2 < 0 ? 0 : costs[i - 1][last2];
19.           }
20.
21.           if (min1 < 0 || costs[i][j] < costs[i][min1]) {
22.             min2 = min1; min1 = j;
23.           } else if (min2 < 0 || costs[i][j] < costs[i][min2]) {
24.             min2 = j;
25.           }
26.         }
27.       }
28.
29.       return costs[n - 1][min1];
30.     }
31.   }
```

Paint House

- Paint House

Paint House

There are a row of n houses, each house can be painted with one of the three colors: red, blue or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by a `n x 3` cost matrix. For example, `costs[0][0]` is the cost of painting house 0 with color red; `costs[1][2]` is the cost of painting house 1 with color green, and so on… Find the minimum cost to paint all houses.

Note:

All costs are positive integers.

Solution:

```
1.  public class Solution {
2.    public int minCost(int[][] costs) {
3.      if (costs == null || costs.length == 0) {
4.        return 0;
5.      }
6.
7.      int n = costs.length;
```

```
 8.
 9.     for (int i = 1; i < n; i++) {
10.        costs[i][0] += Math.min(costs[i - 1][1], costs[i - 1][2]);
11.        costs[i][1] += Math.min(costs[i - 1][0], costs[i - 1][2]);
12.        costs[i][2] += Math.min(costs[i - 1][1], costs[i - 1][0]);
13.     }
14.
15.     return Math.min(Math.min(costs[n - 1][0], costs[n - 1][1]),
    costs[n - 1][2]);
16.   }
17. }
```

Palindrome Partitioning II

- Palindrome Partitioning II

Palindrome Partitioning II

Given a string s, partition s such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of s.

For example, given s = `"aab"` ,
Return `1` since the palindrome partitioning `["aa","b"]` could be produced using 1 cut.

Solution:

```java
public class Solution {
  public int minCut(String s) {
    if (s == null || s.length() == 0) {
      return 0;
    }

    int n = s.length();

    // build the dp matrix to hold the palindrome information
    // dp[i][j] represents whether s[i] to s[j] can form a
    palindrome
    boolean[][] dp = buildMatrix(s, n);

    // res[i] represents the minimum cut needed
    // from s[0] to s[i]
    int[] res = new int[n];

```

```
17.        for (int j = 0; j < n; j++) {
18.            // by default we need j cut from s[0] to s[j]
19.            int cut = j;
20.
21.            for (int i = 0; i <= j; i++) {
22.                if (dp[i][j]) {
23.                    // s[i] to s[j] is a palindrome
24.                    // try to update the cut with res[i - 1]
25.                    cut = Math.min(cut, i == 0 ? 0 : res[i - 1] + 1);
26.                }
27.            }
28.
29.            res[j] = cut;
30.        }
31.
32.        return res[n - 1];
33.    }
34.
35.    boolean[][] buildMatrix(String s, int n) {
36.        boolean[][] dp = new boolean[n][n];
37.
38.        for (int i = n - 1; i >= 0; i--) {
39.            for (int j = i; j < n; j++) {
40.                if (s.charAt(i) == s.charAt(j) && (j - i <= 2 || dp[i + 1]
    [j - 1])) {
41.                    dp[i][j] = true;
42.                }
43.            }
44.        }
45.
46.        return dp;
47.    }
48. }
```

Range Sum Query 2D - Immutable

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).

The above rectangle (with the red border) is defined by (row1, col1) = (2, 1) and (row2, col2) = (4, 3), which contains sum = 8.

Example:

```
1.  Given matrix = [
2.    [3, 0, 1, 4, 2],
3.    [5, 6, 3, 2, 1],
4.    [1, 2, 0, 1, 5],
5.    [4, 1, 0, 1, 7],
6.    [1, 0, 3, 0, 5]
7.  ]
8.
9.  sumRegion(2, 1, 4, 3) -> 8
10. sumRegion(1, 1, 2, 2) -> 11
```

```
11.   sumRegion(1, 2, 2, 4) -> 12
```

Note:

1. You may assume that the matrix does not change.
2. There are many calls to sumRegion function.
3. You may assume that row1 ≤ row2 and col1 ≤ col2.

Solution:

```java
public class NumMatrix {
  int[][] sum;

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

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

    sum = new int[m+1][n+1];

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

  public int sumRegion(int row1, int col1, int row2, int col2) {
    return sum[row2+1][col2+1] - sum[row2+1][col1] - sum[row1]
   [col2+1] + sum[row1][col1];
  }
}


```

```
26.  // Your NumMatrix object will be instantiated and called as such:
27.  // NumMatrix numMatrix = new NumMatrix(matrix);
28.  // numMatrix.sumRegion(0, 1, 2, 3);
29.  // numMatrix.sumRegion(1, 2, 3, 4);
```

Range Sum Query - Immutable

- Range Sum Query - Immutable

Range Sum Query - Immutable

Given an integer array nums, find the sum of the elements between indices i and j (i ≤ j), inclusive.

Example:

```
1.  Given nums = [-2, 0, 3, -5, 2, -1]
2.
3.  sumRange(0, 2) -> 1
4.  sumRange(2, 5) -> -1
5.  sumRange(0, 5) -> -3
```

Note:

1. You may assume that the array does not change.
2. There are many calls to sumRange function.

Solution:

```java
1.  public class NumArray {
2.    int[] sums;
3.
4.    public NumArray(int[] nums) {
5.      sums = new int[nums.length];
6.
7.      for (int i = 0; i < nums.length; i++)
8.        sums[i] = nums[i] + ((i > 0) ? sums[i - 1] : 0);
9.    }
10.
11.    public int sumRange(int i, int j) {
```

```
12.        return (i == 0) ? sums[j] : sums[j] - sums[i - 1];
13.    }
14. }
15.
16.
17. // Your NumArray object will be instantiated and called as such:
18. // NumArray numArray = new NumArray(nums);
19. // numArray.sumRange(0, 1);
20. // numArray.sumRange(1, 2);
```

Russian Doll Envelopes

- Russian Doll Envelopes

Russian Doll Envelopes

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

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

Example:

Given envelopes = `[[5,4],[6,4],[6,7],[2,3]]`, the maximum number of envelopes you can Russian doll is `3` ([2,3] => [5,4] => [6,7]).

Triangle

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.

Solution:

```
public class Solution {
  public int minimumTotal(List<List<Integer>> triangle) {
    int n = triangle.size();

```

```
 5.      int[] arr = new int[n];
 6.
 7.      for (int i = 0; i < n; i++) {
 8.        arr[i] = triangle.get(n - 1).get(i);
 9.      }
10.
11.      for (int i = n - 2; i >= 0; i--) {
12.        for (int j = 0; j <= i; j++) {
13.          arr[j] = Math.min(arr[j], arr[j + 1]) +
     triangle.get(i).get(j);
14.        }
15.      }
16.
17.      return arr[0];
18.    }
19.  }
```

Unique Paths II

- Unique Paths II

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.

Note: m and n will be at most 100.

Solution:

```java
public class Solution {
    public int uniquePathsWithObstacles(int[][] obstacleGrid) {
        int m = obstacleGrid.length;
        int n = obstacleGrid[0].length;
```

```
 5.      int[][] dp = new int[m][n];
 6.
 7.      // first row
 8.      for (int j = 0; j < n; j++) {
 9.        if (obstacleGrid[0][j] == 1) {
10.          break;
11.        }
12.        dp[0][j] = 1;
13.      }
14.
15.      // first col
16.      for (int i = 0; i < m; i++) {
17.        if (obstacleGrid[i][0] == 1) {
18.          break;
19.        }
20.        dp[i][0] = 1;
21.      }
22.
23.      // others
24.      for (int i = 1; i < m; i++) {
25.        for (int j = 1; j < n; j++) {
26.          if (obstacleGrid[i][j] == 0) {
27.            dp[i][j] = dp[i-1][j] + dp[i][j-1];
28.          }
29.        }
30.      }
31.
32.      return dp[m-1][n-1];
33.    }
34.  }
```

Unique Paths

-

Unique Paths

A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot 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 (marked 'Finish' in the diagram below).

How many possible unique paths are there?

Above is a 3 x 7 grid. How many possible unique paths are there?

Note: m and n will be at most 100.

Solution:

```
1.  public class Solution {
2.    public int uniquePaths(int m, int n) {
3.      int[][] dp = new int[m][n];
```

```
 4.
 5.       // first row
 6.       for (int j = 0; j < n; j++) {
 7.         dp[0][j] = 1;
 8.       }
 9.
10.       // first col
11.       for (int i = 0; i < m; i++) {
12.         dp[i][0] = 1;
13.       }
14.
15.       // others
16.       for (int i = 1; i < m; i++) {
17.         for (int j = 1; j < n; j++) {
18.           dp[i][j] = dp[i-1][j] + dp[i][j-1];
19.         }
20.       }
21.
22.       return dp[m-1][n-1];
23.     }
24.  }
```

Word Break II

Word Break II

Given a string s and a dictionary of words dict, add spaces in s to construct a sentence where each word is a valid dictionary word.

Return all such possible sentences.

For example, given

s = `"catsanddog"` ,

dict = `["cat", "cats", "and", "sand", "dog"]` .

A solution is `["cats and dog", "cat sand dog"]` .

Solution:

```
1.  public class Solution {
2.    public List<String> wordBreak(String s, Set<String> dict) {
3.      List<String> res = new ArrayList<String>();
4.
5.      if (!canBreak(s, dict)) return res;
6.
7.      helper(s, dict, 0, "", res);
8.
9.      return res;
10.   }
11.
12.   void helper(String s, Set<String> dict, int start, String sol,
      List<String> res) {
13.     if (start == s.length()) {
14.       res.add(sol);
```

```
15.        return;
16.      }
17.
18.      for (int i = start; i < s.length(); i++) {
19.        String sub = s.substring(start, i + 1);
20.
21.        if (dict.contains(sub)) {
22.          helper(s, dict, i + 1, sol + (sol.length() == 0 ? "" : " ")
     + sub, res);
23.        }
24.      }
25.    }
26.
27.    boolean canBreak(String s, Set<String> dict) {
28.      if (s == null || s.length() == 0) return false;
29.
30.      int n = s.length();
31.
32.      // dp[i] represents whether s[0...i] can be formed by dict
33.      boolean[] dp = new boolean[n];
34.
35.      for (int i = 0; i < n; i++) {
36.        for (int j = 0; j <= i; j++) {
37.          String sub = s.substring(j, i + 1);
38.
39.          if (dict.contains(sub) && (j == 0 || dp[j - 1])) {
40.            dp[i] = true;
41.            break;
42.          }
43.        }
44.      }
45.
46.      return dp[n - 1];
47.    }
48.  }
```

Graph

- Graph

Graph

Clone Graph

- Clone Graph

Clone Graph

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

OJ's undirected graph serialization:
Nodes are labeled uniquely.

We use # as a separator for each node, and , as a separator for node label and each neighbor of the node.
As an example, consider the serialized graph **{0,1,2#1,2#2,2}**.

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

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

Visually, the graph looks like the following:

```
1.       1
2.      / \
```

```
3.        /    \
4.      0 --- 2
5.           / \
6.           \_/
```

Solution:

```
1.  /**
2.   * Definition for undirected graph.
3.   * class UndirectedGraphNode {
4.   *     int label;
5.   *     List<UndirectedGraphNode> neighbors;
6.   *     UndirectedGraphNode(int x) { label = x; neighbors = new
    ArrayList<UndirectedGraphNode>(); }
7.   * };
8.   */
9.  public class Solution {
10.   public UndirectedGraphNode cloneGraph(UndirectedGraphNode root) {
11.     if (root == null) return null;
12.
13.     // use a map to save cloned nodes
14.     Map<UndirectedGraphNode, UndirectedGraphNode> map = new
    HashMap<UndirectedGraphNode, UndirectedGraphNode>();
15.
16.     // clone the root
17.     map.put(root, new UndirectedGraphNode(root.label));
18.
19.     helper(root, map);
20.
21.     return map.get(root);
22.   }
23.
24.   void helper(UndirectedGraphNode root, Map<UndirectedGraphNode,
    UndirectedGraphNode> map) {
25.     for (UndirectedGraphNode neighbor : root.neighbors) {
26.       if (!map.containsKey(neighbor)) {
27.         map.put(neighbor, new UndirectedGraphNode(neighbor.label));
28.         helper(neighbor, map);
```

```
29.          }
30.
31.          map.get(root).neighbors.add(map.get(neighbor));
32.       }
33.    }
34. }
```

Minimum Height Trees

Minimum Height Trees

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

Format

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

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

Example 1:

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

```
1.          0
2.          |
3.          1
```

```
4.          / \
5.         2   3
```

return [1]

Example 2:

Given n = 6 , edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]

```
1.       0  1  2
2.        \ | /
3.          3
4.          |
5.          4
6.          |
7.          5
```

return [3, 4]

Hint:

How many MHTs can a graph have at most?

Note:

1. According to the definition of tree on
   Wikipedia: "a tree is an undirected graph in
   which any two vertices are connected by exactly
   one path. In other words, any connected graph
   without simple cycles is a tree."

2. The height of a rooted tree is the number of
   edges on the longest downward path between the
   root and a leaf.

Reconstruct Itinerary

- Reconstruct Itinerary

Reconstruct Itinerary

Given a list of airline tickets represented by pairs of departure and arrival airports `[from, to]` , reconstruct the itinerary in order. All of the tickets belong to a man who departs from `JFK` . Thus, the itinerary must begin with `JFK` .

Note:

1. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK", "LGB"].
2. All airports are represented by three capital letters (IATA code).
3. You may assume all tickets form at least one valid itinerary.

Example 1:

```
tickets = [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]
```

Return `["JFK", "MUC", "LHR", "SFO", "SJC"]` .

Example 2:

```
tickets = [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],
["ATL","SFO"]]
```

Return `["JFK","ATL","JFK","SFO","ATL","SFO"]` .

Another possible reconstruction is

`["JFK","SFO","ATL","JFK","ATL","SFO"]` . But it is larger in

lexical order.

Solution:

```
1.
2.
```

Greedy

- Greedy

Greedy

Best Time to Buy and Sell Stock II

- Best Time to Buy and Sell Stock II

Best Time to Buy and Sell Stock II

Say you have an array for which the i-th element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times). However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

Solution:

```java
public class Solution {
  public int maxProfit(int[] prices) {
    int profit = 0;

    for (int i = 1; i < prices.length; i++) {
      profit += Math.max(0, prices[i] - prices[i - 1]);
    }

    return profit;
  }
}
```

Time complexity: O(n)

Candy

- Candy

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:

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

What is the minimum candies you must give?

Solution:

```
1.  public class Solution {
2.    public int candy(int[] ratings) {
3.      if (ratings == null || ratings.length == 0) return 0;
4.
5.      int n = ratings.length;
6.
7.      // arr[i] stores the num of candies of i-th kid
8.      int[] arr = new int[n]; arr[0] = 1;
9.
10.     // scan from left to right
11.     for (int i = 1; i < n; i++)
12.       arr[i] = (ratings[i] > ratings[i - 1]) ? arr[i - 1] + 1 : 1;
13.
14.     // scan from right to left
15.     int sum = arr[n - 1];
```

```
16.
17.      for (int i = n - 2; i >= 0; i--) {
18.        if (ratings[i] > ratings[i + 1])
19.          arr[i] = Math.max(arr[i], arr[i + 1] + 1);
20.
21.        sum += arr[i];
22.      }
23.
24.      return sum;
25.    }
26.  }
```

Create Maximum Number

- Create Maximum Number

Create Maximum Number

Given two arrays of length `m` and `n` with digits `0-9` representing two numbers. Create the maximum number of length `k <= m + n` from digits of the two. The relative order of the digits from the same array must be preserved. Return an array of the `k` digits. You should try to optimize your time and space complexity.

Example 1:

nums1 = `[3, 4, 6, 5]`

nums2 = `[9, 1, 2, 5, 8, 3]`

k = `5`

return `[9, 8, 6, 5, 3]`

Example 2:

nums1 = `[6, 7]`

nums2 = `[6, 0, 4]`

k = `5`

return `[6, 7, 6, 0, 4]`

Example 3:

nums1 = `[3, 9]`

nums2 = `[8, 9]`

k = `3`

return `[9, 8, 9]`

Solution:

```
1.
2.
```

Gas Station

- Gas Station

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.

Note:

The solution is guaranteed to be unique.

Solution:

```
1.  public class Solution {
2.    public int canCompleteCircuit(int[] gas, int[] cost) {
3.      int pos = -1;
4.      int curr = 0, total = 0;
5.
6.      for (int i = 0; i < gas.length; i++) {
7.        int diff = gas[i] - cost[i];
8.
9.        curr += diff;
10.       total += diff;
11.
```

```
12.          if (curr < 0) {
13.             curr = 0;
14.             pos = i;
15.          }
16.       }
17.
18.       if (total >= 0) {
19.          return pos + 1;
20.       }
21.
22.       return -1;
23.    }
24. }
```

Jump Game

- Jump Game

Jump Game

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Determine if you are able to reach the last index.

For example:

A = `[2,3,1,1,4]` , return `true` .

A = `[3,2,1,0,4]` , return `false` .

Solution:

```java
1.  public class Solution {
2.    public boolean canJump(int[] nums) {
3.      int reach = 0;
4.
5.      for (int i = 0; i < nums.length && i <= reach; i++) {
6.        if (i + nums[i] > reach) {
7.          reach = i + nums[i];
8.        }
9.      }
10.
11.     return reach >= nums.length - 1;
12.   }
```

```
13.    }
```

Jump Game II

- Jump Game II

Jump Game II

Given an array of non-negative integers, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

For example:
Given array A = `[2,3,1,1,4]`

The minimum number of jumps to reach the last index is `2` . (Jump `1` step from index 0 to 1, then `3` steps to the last index.)

Note:

You can assume that you can always reach the last index.

Solution:

```
1.  public class Solution {
2.    public int jump(int[] nums) {
3.      int reach = 0;
```

```
  4.      int last = 0;
  5.      int steps = 0;
  6.
  7.      for (int i = 0; i < nums.length && i <= reach; i++) {
  8.        if (i > last) {
  9.          steps++;
 10.          last = reach;
 11.        }
 12.
 13.        if (i + nums[i] > reach) {
 14.          reach = i + nums[i];
 15.        }
 16.      }
 17.
 18.      if (reach >= nums.length - 1) {
 19.        return steps;
 20.      } else {
 21.        return -1;
 22.      }
 23.    }
 24.  }
```

Patching Array

Patching Array

Given a sorted positive integer array nums and an integer n, add/patch elements to the array such that any number in range [1, n] inclusive can be formed by the sum of some elements in the array. Return the minimum number of patches required.

Example 1:

nums = `[1, 3]` , n = `6`

Return `1` .

Combinations of nums are `[1], [3], [1,3]` , which form possible sums of: `1, 3, 4` .

Now if we add/patch `2` to nums, the combinations are: `[1], [2], [3], [1,3], [2,3], [1,2,3]` .

Possible sums are `1, 2, 3, 4, 5, 6` , which now covers the range `[1, 6]` .
So we only need `1` patch.

Example 2:

nums = `[1, 5, 10]` , n = `20`

Return `2` .

The two patches can be `[2, 4]` .

Example 3:

nums = `[1, 2, 2]` , n = `5`

Return `0` .

Hash Table

- Hash Table

Hash Table

Binary Tree Vertical Order Traversal

- Binary Tree Vertical Order Traversal

Binary Tree Vertical Order Traversal

Given a binary tree, return the vertical order traversal of its nodes' values. (ie, from top to bottom, column by column).

If two nodes are in the same row and column, the order should be from left to right.

Examples:

```
1.  Given binary tree `[3,9,20,null,null,15,7]`,
2.      3
3.    /\
4.   /  \
5.   9  20
6.      /\
7.     /  \
8.    15   7
```

return its vertical order traversal as:

```
1.  [
2.    [9],
3.    [3,15],
4.    [20],
5.    [7]
6.  ]
```

Given binary tree `[3,9,8,4,0,1,7]` ,

```
1.      3
2.     /\
3.    /  \
4.   9    8
5.  /\   /\
6. /  \/  \
7. 4  01   7
```

return its vertical order traversal as:

```
1. [
2.   [4],
3.   [9],
4.   [3,0,1],
5.   [8],
6.   [7]
7. ]
```

Given binary tree `[3,9,8,4,0,1,7,null,null,null,2,5]` (0's right child is 2 and 1's left child is 5),

```
 1.      3
 2.     /\
 3.    /  \
 4.   9    8
 5.  /\   /\
 6. /  \/  \
 7. 4  01   7
 8.     /\
 9.    /  \
10.   5    2
```

return its vertical order traversal as:

```
1. [
2.   [4],
```

```
3.    [9,5],
4.    [3,0,1],
5.    [8,2],
6.    [7]
7. ]
```

Bulls and Cows

- Bulls and Cows

Bulls and Cows

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

For example:

```
1.  Secret number:  "1807"
2.  Friend's guess: "7810"
```

Hint: `1` bull and `3` cows. (The bull is `8`, the cows are `0`, `1` and `7`.)

Write a function to return a hint according to the secret number and friend's guess, use A to indicate the bulls and B to indicate the cows. In the above example, your function should return `"1A3B"`.

Please note that both secret number and friend's guess may contain duplicate digits, for example:

```
1.  Secret number:  "1123"
2.  Friend's guess: "0111"
```

In this case, the 1st `1` in friend's guess is a bull, the 2nd or 3rd `1` is a cow, and your function should return `"1A1B"`.

You may assume that the secret number and your friend's guess only contain digits, and their lengths are always equal.

Solution:

```
1.  public class Solution {
2.    public String getHint(String secret, String guess) {
3.      int bulls = 0, cows = 0;
4.
5.      int[] nums = new int[10];
6.
7.      for (int i = 0; i < secret.length(); i++) {
8.        int s = secret.charAt(i) - '0';
9.        int g = guess.charAt(i) - '0';
10.
11.       if (s == g) {
12.         bulls++;
13.       } else {
14.         if (nums[s] < 0) {
15.           cows++;
16.         }
17.
18.         if (nums[g] > 0) {
19.           cows++;
20.         }
```

```
21.
22.            nums[s]++;
23.            nums[g]--;
24.         }
25.      }
26.
27.      return bulls + "A" + cows + "B";
28.   }
29. }
```

Contains Duplicate II

- Contains Duplicate II

Contains Duplicate II

Given an array of integers and an integer *k*, find out whether there are two distinct indices *i* and *j* in the array such that **nums[i] = nums[j]** and the difference between *i* and *j* is at most *k*.

Solution:

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

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

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

    return false;
  }
}
```

Contains Duplicate

- Contains Duplicate

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.

Solution:

```
public class Solution {
  public boolean containsDuplicate(int[] nums) {
    Set<Integer> set = new HashSet<>();

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

      if (!set.contains(num)) {
        set.add(num);
      } else {
        return true;
      }
    }

    return false;
  }
}
```

Isomorphic Strings

- Isomorphic Strings

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**.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.

For example,

Given `"egg"`, `"add"`, return true.

Given `"foo"`, `"bar"`, return false.

Given `"paper"`, `"title"`, return true.

Note:

You may assume both **s** and **t** have the same length.

Solution:

```
1.  public class Solution {
2.    public boolean isIsomorphic(String s, String t) {
3.      Map<Character, Character> map1 = new HashMap<>();
```

```
 4.        Map<Character, Character> map2 = new HashMap<>();
 5.
 6.        for (int i = 0; i < s.length(); i++) {
 7.          char c1 = s.charAt(i);
 8.          char c2 = t.charAt(i);
 9.
10.          if (map1.containsKey(c1) && map1.get(c1) != c2)
11.            return false;
12.
13.          if (map2.containsKey(c2) && map2.get(c2) != c1)
14.            return false;
15.
16.          map1.put(c1, c2);
17.          map2.put(c2, c1);
18.        }
19.
20.        return true;
21.    }
22. }
```

Logger Rate Limiter

- Logger Rate Limiter

Logger Rate Limiter

Design a logger system that receive stream of messages along with its timestamps, each message should be printed if and only if it is **not printed in the last 10 seconds**.

Given a message and a timestamp (in seconds granularity), return true if the message should be printed in the given timestamp, otherwise returns false.

It is possible that several messages arrive roughly at the same time.

Example:

```
 1.  Logger logger = new Logger();
 2.
 3.  // logging string "foo" at timestamp 1
 4.  logger.shouldPrintMessage(1, "foo"); returns true;
 5.
 6.  // logging string "bar" at timestamp 2
 7.  logger.shouldPrintMessage(2,"bar"); returns true;
 8.
 9.  // logging string "foo" at timestamp 3
10.  logger.shouldPrintMessage(3,"foo"); returns false;
11.
12.  // logging string "bar" at timestamp 8
13.  logger.shouldPrintMessage(8,"bar"); returns false;
```

```
14.
15.   // logging string "foo" at timestamp 10
16.   logger.shouldPrintMessage(10,"foo"); returns false;
17.
18.   // logging string "foo" at timestamp 11
19.   logger.shouldPrintMessage(11,"foo"); returns true;
```

Maximum Size Subarray Sum Equals k

- Maximum Size Subarray Sum Equals k

Maximum Size Subarray Sum Equals k

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

Example 1:

Given nums = `[1, -1, 5, -2, 3]`, k = `3`,

return `4`. (because the subarray `[1, -1, 5, -2]` sums to 3 and is the longest)

Example 2:

Given nums = `[-2, -1, 2, 1]`, k = `1`,

return `2`. (because the subarray `[-1, 2]` sums to 1 and is the longest)

Follow Up:

Can you do it in O(n) time?

Palindrome Permutation

- Palindrome Permutation

Palindrome Permutation

Given a string, determine if a permutation of the string could form a palindrome.

For example,
`"code"` -> False, `"aab"` -> True, `"carerac"` -> True.

Hint:

1. Consider the palindromes of odd vs even length. What difference do you notice?
2. Count the frequency of each character.
3. If each character occurs even number of times, then it must be a palindrome. How about character which occurs odd number of times?

Solution:

```
1.  public class Solution {
2.    public boolean canPermutePalindrome(String s) {
3.      Set<Character> set = new HashSet<>();
4.
5.      for (int i = 0; i < s.length(); i++) {
6.        char c = s.charAt(i);
7.
8.        if (set.contains(c))
9.          set.remove(c);
10.       else
11.          set.add(c);
```

```
12.        }
13.
14.      return set.size() <= 1;
15.    }
16. }
```

Repeated DNA Sequences

Repeated DNA Sequences

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA.

Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.

For example,

```
1.  Given s = "AAAAACCCCCAAAAACCCCCAAAAAGGGTTT",
2.
3.  Return:
4.  ["AAAAACCCCC", "CCCCCAAAAA"].
```

Solution:

```
1.  public class Solution {
2.    public List<String> findRepeatedDnaSequences(String s) {
3.      int len = s.length();
4.
5.      if (len < 10) {
6.        return new LinkedList<String>();
7.      }
8.
9.      int a = 0;
```

```
10.        int m = 1;
11.
12.        for (int i = 0; i < 10; i++) {
13.          a = a * 4 + getCode(s.charAt(i));
14.          m *= 4;
15.        }
16.
17.        HashSet<Integer> set = new HashSet<Integer>();
18.        HashSet<Integer> r = new HashSet<Integer>();
19.        LinkedList<String> result = new LinkedList<String>();
20.
21.        set.add(a);
22.
23.        int p = 10;
24.
25.        while (p < len) {
26.          a = (a * 4 + getCode(s.charAt(p))) % m;
27.
28.          if (set.contains(a) && !r.contains(a)) {
29.            result.add(s.substring(p - 9, p + 1));
30.            r.add(a);
31.          }
32.          else {
33.            set.add(a);
34.          }
35.
36.          p++;
37.        }
38.
39.        return result;
40.      }
41.
42.      private int getCode(char c) {
43.        if (c == 'C') return 1;
44.        if (c == 'G') return 2;
45.        if (c == 'T') return 3;
46.        return 0;
47.      }
```

```
48.  }
```

Shortest Word Distance II

- Shortest Word Distance II

Shortest Word Distance II

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

Design a class which receives a list of words in the constructor, and implements a method that takes two words word1 and word2 and return the shortest distance between these two words in the list.

For example,

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

Given word1 = `"coding"` , word2 = `"practice"` , return `3` .

Given word1 = `"makes"` , word2 = `"coding"` , return `1` .

Note:

You may assume that word1 does not equal to word2, and word1 and word2 are both in the list.

Solution:

```java
public class WordDistance {
  Map<String, List<Integer>> map;

  public WordDistance(String[] words) {
    map = new HashMap<String, List<Integer>>();

    for (int i = 0; i < words.length; i++) {
      if (!map.containsKey(words[i]))
        map.put(words[i], new ArrayList<Integer>());

      map.get(words[i]).add(i);
    }
  }

  public int shortest(String word1, String word2) {
    List<Integer> l1 = map.get(word1);
    List<Integer> l2 = map.get(word2);

    int i = 0, j = 0, min = Integer.MAX_VALUE;
    int p1 = -1, p2 = -1;

    while (i < l1.size() && j < l2.size()) {
      p1 = l1.get(i);
      p2 = l2.get(j);

      min = Math.min(min, Math.abs(p1 - p2));

      if (p1 < p2) {
        i++;
      } else {
        j++;
      }
    }

    while (i < l1.size()) {
      p1 = l1.get(i++);
      min = Math.min(min, Math.abs(p1 - p2));
    }
```

```
39.
40.      while (j < l2.size()) {
41.        p2 = l2.get(j++);
42.        min = Math.min(min, Math.abs(p1 - p2));
43.      }
44.
45.      return min;
46.    }
47.  }
48.
49.  // Your WordDistance object will be instantiated and called as
       such:
50.  // WordDistance wordDistance = new WordDistance(words);
51.  // wordDistance.shortest("word1", "word2");
52.  // wordDistance.shortest("anotherWord1", "anotherWord2");
```

Single Number

Single Number

Given an array of integers, every element appears twice except for one. Find that single one.

Note:

Your algorithm should have a linear runtime complexity. Could you implement it without using extra memory?

Solution:

```java
public class Solution {
  public int singleNumber(int[] nums) {
    int ans = 0;

    for (int i = 0; i < nums.length; i++)
      ans ^= nums[i];

    return ans;
  }
}
```

Sparse Matrix Multiplication

- Sparse Matrix Multiplication

Sparse Matrix Multiplication

Given two sparse matrices **A** and **B**, return the result of **AB**.

You may assume that **A**'s column number is equal to **B**'s row number.

Example:

```
1.  A = [
2.    [ 1, 0, 0],
3.    [-1, 0, 3]
4.  ]
5.
6.  B = [
7.    [ 7, 0, 0 ],
8.    [ 0, 0, 0 ],
9.    [ 0, 0, 1 ]
10. ]
11.
12.
13.       |  1 0 0 |   | 7 0 0 |   |  7 0 0 |
14.  AB = | -1 0 3 | x | 0 0 0 | = | -7 0 3 |
15.                    | 0 0 1 |
```

Sudoku Solver

- Sudoku Solver

Sudoku Solver

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

Empty cells are indicated by the character `'.` .

You may assume that there will be only one unique solution.

A sudoku puzzle…

…and its solution numbers marked in red.

Solution:

```
1.  public class Solution {
2.    public void solveSudoku(char[][] board) {
3.      if (board == null || board.length != 9 || board[0].length != 9)
   {
4.        return;
5.      }
6.      solve(board, 0, 0);
7.    }
8.
9.    private boolean solve(char[][] board, int i, int j) {
10.     if (j == 9) {
11.       return solve(board, i + 1, 0);
12.     }
13.
14.     if (i == 9) {
15.       return true;
16.     }
17.
18.     if (board[i][j] != '.') {
19.       return solve(board, i, j + 1);
20.     }
21.
22.     // try 1 - 9
```

```
23.       for (int k = 1; k <= 9; k++) {
24.         board[i][j] = (char)('0' + k);
25.
26.         if (check(board, i, j)) {
27.           if (solve(board, i, j + 1)) {
28.             return true;
29.           }
30.         }
31.
32.         board[i][j] = '.';
33.       }
34.
35.       return false;
36.     }
37.
38.     private boolean check(char[][] board, int i, int j) {
39.       // check row i
40.       for (int k = 0; k < 9; k++) {
41.         if (k != j && board[i][k] == board[i][j]) {
42.           return false;
43.         }
44.       }
45.
46.       // check column j
47.       for (int k = 0; k < 9; k++) {
48.         if (k != i && board[k][j] == board[i][j]) {
49.           return false;
50.         }
51.       }
52.
53.       // check grid
54.       int row = i/3 * 3;
55.       int col = j/3 * 3;
56.
57.       for (int r = row; r < row + 3; r++) {
58.         for (int c = col; c < col + 3; c++) {
59.           if (r != i && c != j && board[r][c] == board[i][j]) {
60.             return false;
```

```
61.            }
62.        }
63.      }
64.
65.      return true;
66.    }
67.  }
```

Two Sum III - Data structure design

- Two Sum III - Data structure design

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,

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

Solution:

```
1.  public class TwoSum {
2.    private HashMap<Integer, Integer> map = new HashMap<Integer,
    Integer>();
3.
4.    public void add(int number) {
5.      if (map.containsKey(number)) {
6.        map.put(number, map.get(number) + 1);
7.      } else {
8.        map.put(number, 1);
9.      }
10.   }
11.
```

```
12.    public boolean find(int value) {
13.      for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
14.        int key = entry.getKey();
15.        int idx = entry.getValue();
16.
17.        int val = value - key;
18.
19.        if (key != val && map.containsKey(val)) {
20.          return true;
21.        }
22.
23.        if (key == val && map.get(key) > 1) {
24.          return true;
25.        }
26.      }
27.
28.      return false;
29.    }
30. }
```

Two Sum

- Two Sum

Two Sum

Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input would have exactly one solution.

Example:

```
1.  Given nums = [2, 7, 11, 15], target = 9,
2.
3.  Because nums[0] + nums[1] = 2 + 7 = 9,
4.  return [0, 1].
```

The return format had been changed to zero-based indices. Please read the above updated description carefully.

The idea:

We should be able to find the two numbers by scanning the array once (from left to right). We use a hashmap to store the `value:index` for each number, at `i-th` position, we check if the value of `target - nums[i]` exsists in the hashmap and they have different positions.

Time complexity: `O(n)`

Space complexity: `O(n)`

Solution - Java:

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

    for (int i = 0; i < nums.length; i++) {
      int num1 = nums[i];
      int num2 = target - num1;

      if (map.containsKey(num2)) {
        return new int[]{map.get(num2) + 1, i + 1};
      }

      if (!map.containsKey(num1)) {
        map.put(num1, i);
      }
    }

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

Solution - JavaScript:

```javascript
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function(nums, target) {
  var i, map = {};

```

```
 9.    for (i = 0; i < nums.length; i++) {
10.      var key = target-nums[i];
11.
12.      if (key in map && i != map[key]) {
13.        return [i, map[key]];
14.      }
15.
16.      map[nums[i]] = i;
17.    }
18.
19.    return [-1, -1];
20.  };
```

Unique Word Abbreviation

- Unique Word Abbreviation

Unique Word Abbreviation

An abbreviation of a word follows the form . Below are some examples of word abbreviations:

```
1.  a) it                      --> it    (no abbreviation)
2.
3.       1
4.  b) d|o|g                  --> d1g
5.
6.               1    1 1
7.      1---5----0----5--8
8.  c) i|nternationalizatio|n  --> i18n
9.
10.               1
11.      1---5----0
12.  d) l|ocalizatio|n          --> l10n
```

Assume you have a dictionary and given a word, find whether its abbreviation is unique in the dictionary. A word's abbreviation is unique if no other word from the dictionary has the same abbreviation.

Example:

```
1.  Given dictionary = [ "deer", "door", "cake", "card" ]
2.
3.  isUnique("dear") -> false
4.  isUnique("cart") -> true
```

```
5.  isUnique("cane") -> false
6.  isUnique("make") -> true
```

Solution:

```
1.  public class ValidWordAbbr {
2.    Map<String, Set<String>> map = new HashMap<>();
3.
4.    public ValidWordAbbr(String[] dictionary) {
5.      // build the hashmap
6.      // the key is the abbreviation
7.      // the value is a hash set of the words that have the same
   abbreviation
8.      for (int i = 0; i < dictionary.length; i++) {
9.        String a = abbr(dictionary[i]);
10.       Set<String> set = map.containsKey(a) ? map.get(a) : new
   HashSet<>();
11.       set.add(dictionary[i]);
12.       map.put(a, set);
13.     }
14.   }
15.
16.   public boolean isUnique(String word) {
17.     String a = abbr(word);
18.     // it's unique when the abbreviation does not exist in the map
   or
19.     // it's the only word in the set
20.     return !map.containsKey(a) || (map.get(a).contains(word) &&
   map.get(a).size() == 1);
21.   }
22.
23.   String abbr(String s) {
24.     if (s.length() < 3)
25.       return s;
26.
27.     return s.substring(0, 1) + String.valueOf(s.length() - 2) +
   s.substring(s.length() - 1);
28.   }
```

```
29.  }
30.
31.
32.  // Your ValidWordAbbr object will be instantiated and called as
     such:
33.  // ValidWordAbbr vwa = new ValidWordAbbr(dictionary);
34.  // vwa.isUnique("Word");
35.  // vwa.isUnique("anotherWord");
```

Valid Sudoku

- Valid Sudoku

Valid Sudoku

Determine if a Sudoku is valid, according to: Sudoku Puzzles - The Rules.

The Sudoku board could be partially filled, where empty cells are filled with the character `'.'` .

A partially filled sudoku which is valid.

Note:

A valid Sudoku board (partially filled) is not necessarily solvable. Only the filled cells need to be validated.

Solution:

```
1.  public class Solution {
2.      public boolean isValidSudoku(char[][] board) {
```

```
 3.        int n = board.length, m = board[0].length;
 4.
 5.        if (n != 9 || m != 9)
 6.          return false;
 7.
 8.        // check rows
 9.        for (int i = 0; i < 9; i++) {
10.          Set<Character> set = new HashSet<Character>();
11.
12.          for (int j = 0; j < 9; j++) {
13.            char c = board[i][j];
14.            if (c != '.') {
15.              if (set.contains(c)) return false;
16.              set.add(c);
17.            }
18.          }
19.        }
20.
21.        // check columns
22.        for (int j = 0; j < 9; j++) {
23.          Set<Character> set = new HashSet<Character>();
24.
25.          for (int i = 0; i < 9; i++) {
26.            char c = board[i][j];
27.            if (c != '.') {
28.              if (set.contains(c)) return false;
29.              set.add(c);
30.            }
31.          }
32.        }
33.
34.        // check grids
35.        for (int k = 0; k < 9; k++) {
36.          int row = k / 3 * 3;
37.          int col = k % 3 * 3;
38.
39.          Set<Character> set = new HashSet<Character>();
40.
```

```
41.         for (int i = 0; i < 3; i++) {
42.           for (int j = 0; j < 3; j++) {
43.             char c = board[row + i][col + j];
44.             if (c != '.') {
45.               if (set.contains(c)) return false;
46.               set.add(c);
47.             }
48.           }
49.         }
50.       }
51.
52.       return true;
53.     }
54. }
```

Word Pattern

- Word Pattern

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`.

Examples:

pattern = `"abba"`, str = `"dog cat cat dog"` should return `true`.

pattern = `"abba"`, str = `"dog cat cat fish"` should return `false`.

pattern = `"aaaa"`, str = `"dog cat cat dog"` should return `false`.

pattern = `"abba"`, str = `"dog dog dog dog"` should return `false`.

Notes:

You may assume `pattern` contains only lowercase letters, and `str` contains lowercase letters separated by a single space.

Solution:

```java
public class Solution {
  public boolean wordPattern(String pattern, String str) {
    String[] words = str.split(" ");

    if (pattern.length() != words.length) {
      return false;
    }

    Set<String> set = new HashSet<>();
    Map<Character, String> map = new HashMap<>();

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

      if (map.containsKey(p) && !map.get(p).equals(s)) {
        return false;
      }

      if (!map.containsKey(p) && set.contains(s)) {
        return false;
      }

      map.put(p, s);
      set.add(s);
    }

    return true;
  }
}
```

Heap

Heap

Design Twitter

- Design Twitter

Design Twitter

Design a simplified version of Twitter where users can post tweets, follow/unfollow another user and is able to see the 10 most recent tweets in the user's news feed. Your design should support the following methods:

1. **postTweet(userId, tweetId)**: Compose a new tweet.

2. **getNewsFeed(userId)**: Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user herself. Tweets must be ordered from most recent to least recent.

3. **follow(followerId, followeeId)**: Follower follows a followee.

4. **unfollow(followerId, followeeId)**: Follower unfollows a followee.

Example:

```
1.  Twitter twitter = new Twitter();
2.
3.  // User 1 posts a new tweet (id = 5).
4.  twitter.postTweet(1, 5);
5.
```

```
 6.  // User 1's news feed should return a list with 1 tweet id -> [5].
 7.  twitter.getNewsFeed(1);
 8.
 9.  // User 1 follows user 2.
10.  twitter.follow(1, 2);
11.
12.  // User 2 posts a new tweet (id = 6).
13.  twitter.postTweet(2, 6);
14.
15.  // User 1's news feed should return a list with 2 tweet ids -> [6,
     5].
16.  // Tweet id 6 should precede tweet id 5 because it is posted after
     tweet id 5.
17.  twitter.getNewsFeed(1);
18.
19.  // User 1 unfollows user 2.
20.  twitter.unfollow(1, 2);
21.
22.  // User 1's news feed should return a list with 1 tweet id -> [5],
23.  // since user 1 is no longer following user 2.
24.  twitter.getNewsFeed(1);
```

Find Median from Data Stream

- Find Median from Data Stream

Find Median from Data Stream

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

Examples:

`[2,3,4]` , the median is `3`

`[2,3]` , the median is `(2 + 3) / 2 = 2.5`

Design a data structure that supports the following two operations:

- void addNum(int num) - Add a integer number from the data stream to the data structure.
- double findMedian() - Return the median of all elements so far.

For example:

```
1. add(1)
2. add(2)
3. findMedian() -> 1.5
4. add(3)
5. findMedian() -> 2
```

Solution:

```java
class MedianFinder {
  // Max heap
  PriorityQueue<Integer> left = new PriorityQueue<>(new
Comparator<Integer>() {
    public int compare(Integer a, Integer b) { return b - a; }
  });

  // Min heap
  PriorityQueue<Integer> right = new PriorityQueue<>(new
Comparator<Integer>() {
    public int compare(Integer a, Integer b) { return a - b; }
  });

  // Adds a number into the data structure.
  public void addNum(int num) {
    double m = findMedian();

    int diff = left.size() - right.size();

    // left and right are balanced
    if (diff == 0) {
      if (num < m) {
        left.offer(num);
      } else {
        right.offer(num);
      }
    }
    // left has more elements than right
    else if (diff > 0) {
      if (num < m) {
        if (left.size() > 0) right.offer(left.poll());
        left.offer(num);
      } else {
        right.offer(num);
      }
    }
```

```
35.        // right has more elements than left
36.        else {
37.          if (num < m) {
38.            left.offer(num);
39.          } else {
40.            if (right.size() > 0) left.offer(right.poll());
41.            right.offer(num);
42.          }
43.        }
44.      }
45.
46.      // Returns the median of current data stream
47.      public double findMedian() {
48.        if (left.size() == 0 && right.size() == 0) {
49.          return 0.0;
50.        }
51.
52.        int diff = left.size() - right.size();
53.
54.        // left and right are balanced
55.        if (diff == 0) {
56.          return (left.peek() + right.peek()) / 2.0;
57.        }
58.        // left has more elements than right
59.        else if (diff > 0) {
60.          return left.peek();
61.        }
62.        // right has more elements than left
63.        else {
64.          return right.peek();
65.        }
66.      }
67.    };
68.
69.    // Your MedianFinder object will be instantiated and called as
       such:
70.    // MedianFinder mf = new MedianFinder();
71.    // mf.addNum(1);
```

```
72.   // mf.findMedian();
```

Kth Largest Element in an Array

- Kth Largest Element in an Array

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 ≤ k ≤ array's length.

Solution:

```java
public class Solution {
  public int findKthLargest(int[] a, int k) {
    int n = a.length;
    int p = quickSelect(a, 0, n - 1, n - k + 1);
    return a[p];
  }

  // return the index of the kth smallest number
  int quickSelect(int[] a, int lo, int hi, int k) {
    // use quick sort's idea
    // put nums that are <= pivot to the left
    // put nums that are  > pivot to the right
    int i = lo, j = hi, pivot = a[hi];
    while (i < j) {
```

```
15.            if (a[i++] > pivot)
16.                swap(a, --i, --j);
17.          }
18.          swap(a, i, hi);
19.
20.          // count the nums that are <= pivot from lo
21.          int m = i - lo + 1;
22.
23.          // pivot is the one!
24.          if (m == k)      return i;
25.          // pivot is too big, so it must be on the left
26.          else if (m > k) return quickSelect(a, lo, i - 1, k);
27.          // pivot is too small, so it must be on the right
28.          else            return quickSelect(a, i + 1, hi, k - m);
29.      }
30.
31.      void swap(int[] a, int i, int j) {
32.          int tmp = a[i];
33.          a[i] = a[j];
34.          a[j] = tmp;
35.      }
36.  }
```

Rearrange String k Distance Apart

- Rearrange String k Distance Apart

Rearrange String k Distance Apart

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

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

Example 1:

```
1.  str = "aabbcc", k = 3
2.
3.  Result: "abcabc"
4.
5.  The same letters are at least distance 3 from each other.
```

Example 2:

```
1.  str = "aaabc", k = 3
2.
3.  Answer: ""
4.
5.  It is not possible to rearrange the string.
```

Example 3:

```
1.  str = "aaadbbcc", k = 2
2.
```

```
3.  Answer: "abacabcd"
4.
5.  Another possible answer is: "abcabcda"
6.
7.  The same letters are at least distance 2 from each other.
```

Sliding Window Maximum

Sliding Window Maximum

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

For example,

Given nums = `[1,3,-1,-3,5,3,6,7]`, and `k = 3`.

```
1.  Window position               Max
2.  ---------------               -----
3.  [1  3  -1] -3  5  3  6  7       3
4.   1 [3  -1  -3] 5  3  6  7       3
5.   1  3 [-1  -3  5] 3  6  7       5
6.   1  3  -1 [-3  5  3] 6  7       5
7.   1  3  -1  -3 [5  3  6] 7       6
8.   1  3  -1  -3  5 [3  6  7]      7
```

Therefore, return the max sliding window as `[3,3,5,5,6,7]`.

Note:

You may assume k is always valid, ie: 1 ≤ k ≤ input array's size for non-empty array.

Follow up:

Could you solve it in linear time?

Hint:

1. How about using a data structure such as deque
   (double-ended queue)?
2. The queue size need not be the same as the
   window's size.
3. Remove redundant elements and the queue should
   store only elements that need to be considered.

Solution:

```
1.   public class Solution {
2.     public int[] maxSlidingWindow(int[] nums, int w) {
3.       if (nums == null || nums.length == 0 || w < 1)
4.         return new int[0];
5.
6.       int n = nums.length, k = 0;
7.       int[] res = new int[n - w + 1];
8.
9.       // use a deque to control the window
10.      Deque<Integer> q = new ArrayDeque<Integer>();
11.
12.      for (int i = 0; i < n; i++) {
13.        // exceeds window size
14.        if (!q.isEmpty() && q.peekFirst() + w <= i)
15.          q.removeFirst();
16.
17.        // pop those smaller ones from behind
18.        while (!q.isEmpty() && nums[q.peekLast()] <= nums[i])
19.          q.removeLast();
20.
21.        q.addLast(i);
22.
23.        if (i >= w - 1)
```

```
24.          res[k++] = nums[q.peekFirst()];
25.      }
26.
27.      return res;
28.   }
29. }
```

Top K Frequent Elements

- Top K Frequent Elements

Top K Frequent Elements

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

For example,
Given `[1,1,1,2,2,3]` and k = `2` , return `[1,2]` .

Note:

- You may assume k is always valid, 1 ≤ k ≤ number of unique elements.
- Your algorithm's time complexity **must be** better than O(n log n), where n is the array's size.

Linked List

- Linked List

Linked List

Convert Sorted List to Binary Search Tree

Convert Sorted List to Binary Search Tree

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public TreeNode sortedListToBST(ListNode head) {
    if (head == null) {
      return null;
    }
```

```
23.
24.      if (head.next == null) {
25.        TreeNode root = new TreeNode(head.val);
26.        return root;
27.      }
28.
29.      // find middle node
30.      ListNode prev = null;
31.      ListNode slow = head;
32.      ListNode fast = head;
33.
34.      while (fast != null && fast.next != null) {
35.        prev = slow;
36.        slow = slow.next;
37.        fast = fast.next.next;
38.      }
39.
40.      // slow is the middle node
41.      TreeNode root = new TreeNode(slow.val);
42.
43.      prev.next = null;
44.
45.      root.left = sortedListToBST(head);
46.      root.right = sortedListToBST(slow.next);
47.
48.      return root;
49.    }
50.  }
```

Copy List with Random Pointer

- Copy List with Random Pointer

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.

Solution:

```
1.   /**
2.    * Definition for singly-linked list with a random pointer.
3.    * class RandomListNode {
4.    *      int label;
5.    *      RandomListNode next, random;
6.    *      RandomListNode(int x) { this.label = x; }
7.    * };
8.    */
9.   public class Solution {
10.     public RandomListNode copyRandomList(RandomListNode head) {
11.       if (head == null) return null;
12.
13.       Map<RandomListNode, RandomListNode> map = new
      HashMap<RandomListNode, RandomListNode>();
14.
15.       // step 1. copy all the nodes
16.       RandomListNode node = head;
17.       while (node != null) {
18.         map.put(node, new RandomListNode(node.label));
19.         node = node.next;
20.       }
21.
```

```
22.        // step 2. assign next and random pointers
23.        node = head;
24.        while (node != null) {
25.          map.get(node).next = map.get(node.next);
26.          map.get(node).random = map.get(node.random);
27.          node = node.next;
28.        }
29.
30.        return map.get(head);
31.    }
32. }
```

Delete Node in a Linked List

- Delete Node in a Linked List

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.

Solution:

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

Intersection of Two Linked Lists

- Intersection of Two Linked Lists

Intersection of Two Linked Lists

Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:

```
1.  A:          a1 → a2
2.                    ↘
3.                      c1 → c2 → c3
4.                    ↗
5.  B:     b1 → b2 → b3
```

begin to intersect at node c1.

Notes:

- If the two linked lists have no intersection at all, return null.
- The linked lists must retain their original structure after the function returns.
- You may assume there are no cycles anywhere in the entire linked structure.
- Your code should preferably run in O(n) time and use only O(1) memory.

Solution:

```
1.  /**
2.   * Definition for singly-linked list.
3.   * public class ListNode {
4.   *     int val;
5.   *     ListNode next;
```

```
 6.  *     ListNode(int x) {
 7.  *         val = x;
 8.  *         next = null;
 9.  *     }
10.  * }
11.  */
12.  public class Solution {
13.    public ListNode getIntersectionNode(ListNode l1, ListNode l2) {
14.      if (l1 == null || l2 == null) return null;
15.
16.      // step 1. count the two lists
17.      int n1 = count(l1), n2 = count(l2);
18.
19.      // step 2. move the longer one |n2 - n1| steps
20.      int n = Math.abs(n1 - n2);
21.
22.      while (n-- > 0) {
23.        if (n1 > n2)
24.          l1 = l1.next;
25.        else
26.          l2 = l2.next;
27.      }
28.
29.      // step 3. move together and find the meeting point
30.      while (l1 != l2) {
31.        l1 = l1.next;
32.        l2 = l2.next;
33.      }
34.
35.      return l1;
36.    }
37.
38.    int count(ListNode l) {
39.      if (l == null) return 0;
40.      return 1 + count(l.next);
41.    }
42.  }
```

Merge k Sorted Lists

- Merge k Sorted Lists

Merge k Sorted Lists

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

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        ListNode f = new ListNode(0);
        ListNode p = f;

        int k = lists.length;

        PriorityQueue<ListNode> heap = new PriorityQueue<>(k + 1, new
    Comparator<ListNode>() {
            public int compare(ListNode a, ListNode b) {
                return a.val - b.val;
            }
        });

        // step 1. put the first node of every list to the min heap
        for (int i = 0; i < k; i++) {
            if (lists[i] != null) {
```

```
25.          heap.offer(lists[i]);
26.        }
27.      }
28.
29.      while (!heap.isEmpty()) {
30.        ListNode node = heap.poll();
31.
32.        p.next = node;
33.        p = p.next;
34.
35.        if (node.next != null) {
36.          heap.offer(node.next);
37.        }
38.      }
39.
40.      p.next = null;
41.      return f.next;
42.    }
43. }
```

Merge Two Sorted Lists

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.

Solution:

```java
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode l = new ListNode(0);
        ListNode p = l;

        while (l1 != null || l2 != null) {
            if (l1 == null) {
                p.next = l2;
                break;
            }

            if (l2 == null) {
                p.next = l1;
                break;
            }

```

```
25.         if (l1.val < l2.val) {
26.             p.next = l1;
27.             l1 = l1.next;
28.         } else {
29.             p.next = l2;
30.             l2 = l2.next;
31.         }
32.
33.         p = p.next;
34.     }
35.
36.     return l.next;
37.   }
38. }
```

Odd Even Linked List

-

Odd Even Linked List

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.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

Example:

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

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

Note:

- The relative order inside both the even and odd groups should remain as it was in the input.
- The first node is considered odd, the second node even and so on …

Plus One Linked List

- Plus One Linked List

Plus One Linked List

Given a non-negative number represented as a singly linked list of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

Example:

```
1.  Input:
2.  1->2->3
3.
4.  Output:
5.  1->2->4
```

Remove Duplicates from Sorted List

- Remove Duplicates from Sorted List

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`.

Solution:

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

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

```

```
20.        return head;
21.    }
22. }
```

Remove Duplicates from Sorted List II

- Remove Duplicates from Sorted List II

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

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

Solution:

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

    ListNode fake = new ListNode(0);
    ListNode p = fake;
    ListNode slow = head;
    ListNode fast = head;
```

```
19.
20.      while (fast != null) {
21.        int count = 0;
22.
23.        while (fast != null && fast.val == slow.val) {
24.          count++;
25.          fast = fast.next;
26.        }
27.
28.        if (count == 1) {
29.          p.next = slow;
30.          p = p.next;
31.        }
32.
33.        slow = fast;
34.      }
35.
36.      p.next = null;
37.
38.      return fake.next;
39.    }
40.  }
```

Remove Linked List Elements

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`

Solution:

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

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

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

```
20.            p = p.next;
21.        }
22.         head = head.next;
23.      }
24.
25.     p.next = null;
26.
27.     return f.next;
28.   }
29. }
```

Reorder List

- Reorder List

Reorder List

Given a singly linked list L: $L_0 \to L_1 \to \ldots \to L_{n-1} \to L_n$,

reorder it to: $L_0 \to L_n \to L_1 \to L_{n-1} \to L_2 \to L_{n-2} \to \ldots$

You must do this in-place without altering the nodes' values.

For example,

Given `{1,2,3,4}`, reorder it to `{1,4,2,3}`.

Solution:

```
1.  /**
2.   * Definition for singly-linked list.
3.   * class ListNode {
4.   *     int val;
5.   *     ListNode next;
6.   *     ListNode(int x) {
7.   *         val = x;
8.   *         next = null;
9.   *     }
10.  * }
11.  */
12.  public class Solution {
13.    public void reorderList(ListNode head) {
14.      if (head == null || head.next == null)
15.        return;
16.
17.      // step 1. cut the list to two halves
```

```
18.        // prev will be the tail of 1st half
19.        // slow will be the head of 2nd half
20.        ListNode prev = null, slow = head, fast = head, l1 = head;
21.
22.        while (fast != null && fast.next != null) {
23.          prev = slow;
24.          slow = slow.next;
25.          fast = fast.next.next;
26.        }
27.
28.        prev.next = null;
29.
30.        // step 2. reverse the 2nd half
31.        ListNode l2 = reverse(slow);
32.
33.        // step 3. merge the two halves
34.        merge(l1, l2);
35.      }
36.
37.      ListNode reverse(ListNode head) {
38.        ListNode prev = null, curr = head, next = null;
39.
40.        while (curr != null) {
41.          next = curr.next;
42.          curr.next = prev;
43.          prev = curr;
44.          curr = next;
45.        }
46.
47.        return prev;
48.      }
49.
50.      void merge(ListNode l1, ListNode l2) {
51.        while (l1 != null) {
52.          ListNode n1 = l1.next, n2 = l2.next;
53.          l1.next = l2;
54.
55.          if (n1 == null)
```

```
56.         break;
57.
58.      l2.next = n1;
59.      l1 = n1;
60.      l2 = n2;
61.    }
62.   }
63. }
```

Reverse Linked List

Reverse Linked List

Reverse a singly linked list.

Hint:

A linked list can be reversed either iteratively or recursively. Could you implement both?

Solution:

```
 1.  /**
 2.   * Definition for singly-linked list.
 3.   * public class ListNode {
 4.   *     int val;
 5.   *     ListNode next;
 6.   *     ListNode(int x) { val = x; }
 7.   * }
 8.   */
 9.  public class Solution {
10.    public ListNode reverseList(ListNode head) {
11.      if (head == null)
12.        return null;
13.
14.      ListNode prev = null, curr = head, next = null;
15.
16.      while (curr != null) {
17.        next = curr.next;
18.        curr.next = prev;
19.        prev = curr;
20.        curr = next;
21.      }
```

```
22.
23.        return prev;
24.    }
25. }
```

Reverse Linked List II

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`.

Note:

Given m, n satisfy the following condition: 1 ≤ m ≤ n ≤ length of list.

Solution:

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

```

```
15.        int k = n - m + 1;
16.
17.        ListNode f = new ListNode(0);
18.        f.next = head;
19.
20.        ListNode p = f;
21.        ListNode c = head;
22.
23.        // locate the m-th node
24.        while (m > 1) {
25.          p = c;
26.          c = c.next;
27.          m--;
28.        }
29.
30.        // reverse till n-th node
31.        ListNode prev = null;
32.        ListNode curr = c;
33.        ListNode next = null;
34.
35.        while (k > 0) {
36.          next = curr.next;
37.          curr.next = prev;
38.          prev = curr;
39.          curr = next;
40.          k--;
41.        }
42.
43.        // re-link
44.        p.next = prev;
45.        c.next = curr;
46.
47.        return f.next;
48.    }
49. }
```

Reverse Nodes in k-Group

- Reverse Nodes in k-Group

Reverse Nodes in k-Group

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.

You may not alter the values in the nodes, only nodes itself may be changed.

Only constant memory is allowed.

For example,

Given this linked list: `1->2->3->4->5`

For k = 2, you should return: `2->1->4->3->5`

For k = 3, you should return: `3->2->1->4->5`

Solution:

```
1.  /**
2.   * Definition for singly-linked list.
3.   * public class ListNode {
4.   *     int val;
5.   *     ListNode next;
6.   *     ListNode(int x) { val = x; }
7.   * }
8.   */
```

```java
 9.  public class Solution {
10.    public ListNode reverseKGroup(ListNode head, int k) {
11.      if (getLength(head) < k) {
12.        return head;
13.      }
14.
15.      ListNode prev = null;
16.      ListNode curr = head;
17.
18.      int count = k;
19.      while (curr != null && count > 0) {
20.        ListNode next = curr.next;
21.        curr.next = prev;
22.        prev = curr;
23.        curr = next;
24.        count--;
25.      }
26.
27.      // prev is the new head
28.      // head is the new tail
29.      // curr is the next list
30.
31.      head.next = reverseKGroup(curr, k);
32.
33.      return prev;
34.    }
35.
36.    int getLength(ListNode head) {
37.        int len = 0;
38.
39.      while (head != null) {
40.            head = head.next;
41.            len++;
42.      }
43.
44.      return len;
45.    }
46.  }
```

Swap Nodes in Pairs

- Swap Nodes in Pairs

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.

Solution:

```
 1.  /**
 2.   * Definition for singly-linked list.
 3.   * public class ListNode {
 4.   *     int val;
 5.   *     ListNode next;
 6.   *     ListNode(int x) {
 7.   *         val = x;
 8.   *         next = null;
 9.   *     }
10.   * }
11.   */
12.  public class Solution {
13.    public ListNode swapPairs(ListNode head) {
14.      // Base Case: The list is empty or has only one node
15.      if (head == null || head.next == null)
```

```
16.        return head;
17.
18.     // Store head of list after two nodes
19.     ListNode rest = head.next.next;
20.
21.     // Change head
22.     ListNode newhead = head.next;
23.
24.     // Change next of second node
25.     head.next.next = head;
26.
27.     // Recur for remaining list and change next of head
28.     head.next = swapPairs(rest);
29.
30.     // Return new head of modified list
31.     return newhead;
32.   }
33. }
```

Math

- Math

Math

Add Digits

- Add Digits

Add Digits

Given a non-negative integer `num`, repeatedly add all its digits until the result has only one digit.

For example:

Given `num = 38`, the process is like: `3 + 8 = 11`, `1 + 1 = 2`. Since `2` has only one digit, return it.

Follow up:

Could you do it without any loop/recursion in O(1) runtime?

Hint:

1. A naive implementation of the above process is trivial. Could you come up with other methods?
2. What are all the possible results?
3. How do they occur, periodically or randomly?
4. You may find this Wikipedia article useful.

Solution:

```
public class Solution {
  public int addDigits(int num) {
    return 1 + (num - 1) % 9;
  }
}
```

Add Two Numbers

- Add Two Numbers

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

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
  public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode f = new ListNode(0);
    ListNode p = f;

    int c = 0;

    while (l1 != null || l2 != null || c != 0) {
      int a = (l1 == null) ? 0 : l1.val;
```

```
18.          int b = (l2 == null) ? 0 : l2.val;
19.          int s = a + b + c;
20.
21.          p.next = new ListNode(s % 10);
22.          p = p.next;
23.
24.          if (l1 != null) l1 = l1.next;
25.          if (l2 != null) l2 = l2.next;
26.
27.          c = s / 10;
28.        }
29.
30.        return f.next;
31.      }
32.  }
```

Bulb Switcher

- Bulb Switcher

Bulb Switcher

There are n bulbs that are initially off. You first turn on all the bulbs. Then, you turn off every second bulb. On the third round, you toggle every third bulb (turning on if it's off or turning off if it's on). For the ith round, you toggle every i bulb. For the nth round, you only toggle the last bulb. Find how many bulbs are on after n rounds.

Example:

```
1.  Given n = 3.
2.
3.  At first, the three bulbs are [off, off, off].
4.  After first round, the three bulbs are [on, on, on].
5.  After second round, the three bulbs are [on, off, on].
6.  After third round, the three bulbs are [on, off, off].
7.
8.  So you should return 1, because there is only one bulb is on.
```

Count Numbers with Unique Digits

Count Numbers with Unique Digits

Given a **non-negative** integer n, count all numbers with unique digits, x, where 0 ≤ x < 10^n.

Example:

Given n = 2, return 91. (The answer should be the total numbers in the range of 0 ≤ x < 100, excluding `[11,22,33,44,55,66,77,88,99]`)

Hint:

1. A direct way is to use the backtracking approach.
2. Backtracking should contains three states which are (the current number, number of steps to get that number and a bitmask which represent which number is marked as visited so far in the current number). Start with state (0,0,0) and count all valid number till we reach number of steps equals to 10n.
3. This problem can also be solved using a dynamic programming approach and some knowledge of combinatorics.
4. Let f(k) = count of numbers with unique digits with length equals k.

5. f(1) = 10, …, f(k) = 9 *9* 8 * … (9 - k + 2) [The first factor is 9 because a number cannot start with 0].

Count Primes

- Count Primes

Count Primes

Description:

Count the number of prime numbers less than a non-negative number, n.

Hint:

- Let's start with a isPrime function. To determine if a number is prime, we need to check if it is not divisible by any number less than n. The runtime complexity of isPrime function would be O(n) and hence counting the total prime numbers up to n would be O(n2). Could we do better?

- As we know the number must not be divisible by any number > n / 2, we can immediately cut the total iterations half by dividing only up to n / 2. Could we still do better?

- Let's write down all of 12's factors:

  ```
  1.  2 × 6 = 12
  2.  3 × 4 = 12
  3.  4 × 3 = 12
  4.  6 × 2 = 12
  ```

As you can see, calculations of 4 × 3 and 6 × 2 are not necessary. Therefore, we only need to consider factors up to √n because, if n is divisible by some number p, then n = p × q and since p ≤ q, we could derive that p ≤ √n. Our total runtime has now improved to $O(n^{1.5})$, which is slightly better. Is there a faster approach?

```java
public int countPrimes(int n) {
    int count = 0;
    for (int i = 1; i < n; i++) {
        if (isPrime(i)) count++;
    }
    return count;
}

private boolean isPrime(int num) {
    if (num <= 1) return false;
    // Loop's ending condition is i * i <= num instead of i <= sqrt(num)
    // to avoid repeatedly calling an expensive function sqrt().
    for (int i = 2; i * i <= num; i++) {
        if (num % i == 0) return false;
    }
    return true;
}
```

- The Sieve of Eratosthenes is one of the most efficient ways to find all prime numbers up to n. But don't let that name scare you, I promise that the concept is surprisingly simple.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

Sieve of Eratosthenes: algorithm steps for primes below 121. "Sieve of Eratosthenes Animation" by SKopp is licensed under CC BY 2.0.

We start off with a table of n numbers. Let's look at the first number, 2. We know all multiples of 2 must not be primes, so we mark them off as non-primes. Then we look at the next number, 3. Similarly, all multiples of 3 such as 3 × 2 = 6, 3 × 3 = 9, … must not be primes, so we mark them off as well. Now we look at the next number, 4, which was already marked off. What does this tell you? Should you mark off all multiples of 4 as well?

- 4 is not a prime because it is divisible by 2, which means all multiples of 4 must also be

divisible by 2 and were already marked off. So we can skip 4 immediately and go to the next number, 5. Now, all multiples of 5 such as 5 × 2 = 10, 5 × 3 = 15, 5 × 4 = 20, 5 × 5 = 25, … can be marked off. There is a slight optimization here, we do not need to start from 5 × 2 = 10. Where should we start marking off?

- In fact, we can mark off multiples of 5 starting at 5 × 5 = 25, because 5 × 2 = 10 was already marked off by multiple of 2, similarly 5 × 3 = 15 was already marked off by multiple of 3. Therefore, if the current number is p, we can always mark off multiples of p starting at p2, then in increments of p: p2 + p, p2 + 2p, … Now what should be the terminating loop condition?

- It is easy to say that the terminating loop condition is p < n, which is certainly correct but not efficient. Do you still remember Hint #3?

- Yes, the terminating loop condition can be p < √n, as all non-primes ≥ √n must have already been marked off. When the loop terminates, all the numbers in the table that are non-marked are prime.

  The Sieve of Eratosthenes uses an extra O(n) memory and its runtime complexity is O(n log log

n). For the more mathematically inclined readers, you can read more about its algorithm complexity on Wikipedia.

```java
public int countPrimes(int n) {
    boolean[] isPrime = new boolean[n];
    for (int i = 2; i < n; i++) {
        isPrime[i] = true;
    }
    // Loop's ending condition is i * i < n instead of i < sqrt(n)
    // to avoid repeatedly calling an expensive function sqrt().
    for (int i = 2; i * i < n; i++) {
        if (!isPrime[i]) continue;
        for (int j = i * i; j < n; j += i) {
            isPrime[j] = false;
        }
    }
    int count = 0;
    for (int i = 2; i < n; i++) {
        if (isPrime[i]) count++;
    }
    return count;
}
```

Divide Two Integers

- Divide Two Integers

Divide Two Integers

Divide two integers without using multiplication, division and mod operator.

If it is overflow, return MAX_INT.

Solution:

```java
public class Solution {
  public int divide(int dividend, int divisor) {
    if (dividend == Integer.MIN_VALUE && divisor == -1) {
      return Integer.MAX_VALUE;
    }

    long p = Math.abs((long)dividend);
    long q = Math.abs((long)divisor);

    int res = 0;

    while (p >= q) {
      int count = 0;
      while (p >= (q << count)) {
        count++;
      }

      p -= q << (count - 1);
      res += 1 << (count - 1);
    }

    return ((dividend^divisor) >>> 31 == 0) ? res : -res;
  }
```

```
24.  }
```

Excel Sheet Column Number

- Excel Sheet Column Number

Excel Sheet Column Number

Related to question Excel Sheet Column Title

Given a column title as appear in an Excel sheet, return its corresponding column number.

For example:

```
1.     A -> 1
2.     B -> 2
3.     C -> 3
4.     ...
5.     Z -> 26
6.     AA -> 27
7.     AB -> 28
```

Solution:

```java
public class Solution {
  public int titleToNumber(String s) {
    int num = 0;

    for (int i = 0; i < s.length(); i++) {
      num = num * 26 + (int)(s.charAt(i) - 'A') + 1;
    }

    return num;
  }
}
```

Excel Sheet Column Title

- Excel Sheet Column Title

Excel Sheet Column Title

Given a positive integer, return its corresponding column title as appear in an Excel sheet.

For example:

```
1 -> A
2 -> B
3 -> C
...
26 -> Z
27 -> AA
28 -> AB
```

Solution:

```java
public class Solution {
  public String convertToTitle(int n) {
    String res = "";

    while (n >= 1) {
      res = (char)('A' + (n - 1) % 26) + res;
      n = (n - 1) / 26;
    }

    return res;
  }
}
```

Factorial Trailing Zeroes

- Factorial Trailing Zeroes

Factorial Trailing Zeroes

Given an integer n, return the number of trailing zeroes in n!.

Note: Your solution should be in logarithmic time complexity.

Solution:

```
public class Solution {
  public int trailingZeroes(int n) {
    int res = 0;

    while (n > 0) {
      n /= 5;
      res += n;
    }

    return res;
  }
}
```

Fraction to Recurring Decimal

- Fraction to Recurring Decimal

Fraction to Recurring Decimal

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

For example,

- Given numerator = 1, denominator = 2, return "0.5".
- Given numerator = 2, denominator = 1, return "2".
- Given numerator = 2, denominator = 3, return "0.(6)".

Hint:

1. No scary math, just apply elementary math knowledge. Still remember how to perform a long division?
2. Try a long division on 4/9, the repeating part is obvious. Now try 4/333. Do you see a pattern?
3. Be wary of edge cases! List out as many test cases as you can think of and test your code

thoroughly.

Solution:

```java
public class Solution {
  public String fractionToDecimal(int numerator, int denominator) {
    // zero denominator
    if (denominator == 0) return "NaN";

    // zero numerator
    if (numerator == 0) return "0";

    StringBuilder res = new StringBuilder();

    Long n = new Long(numerator);
    Long d = new Long(denominator);

    // determine the sign
    if (n < 0 ^ d < 0) res.append('-');

    // remove sign of operands
    n = Math.abs(n); d = Math.abs(d);

    // append integral part
    res.append(Long.toString(n / d));

    // in case no fractional part
    if (n % d == 0) return res.toString();

    res.append('.');

    Map<Long, Integer> map = new HashMap<Long, Integer>();

    // simulate the division process
    for (Long r = n % d; r > 0; r %= d) {
      // meet a known remainder
      // so we reach the end of the repeating part
      if (map.containsKey(r)) {
```

```
35.          res.insert(map.get(r), "(");
36.          res.append(")");
37.          break;
38.        }
39.
40.        // the remainder is first seen
41.        // remember the current position for it
42.        map.put(r, res.length());
43.
44.        r *= 10;
45.
46.        // append the quotient digit
47.        res.append(Long.toString(r / d));
48.      }
49.
50.      return res.toString();
51.    }
52.  }
```

Happy Number

- Happy Number

Happy Number

Write an algorithm to determine if a number is "happy".

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.

Example: 19 is a happy number

1^2 + 9^2 = 82

8^2 + 2^2 = 68

6^2 + 8^2 = 100

1^2 + 0^2 + 0^2 = 1

Solution:

```
1.  public class Solution {
2.    public boolean isHappy(int n) {
3.      Set<Integer> set = new HashSet<Integer>();
4.
5.      while (n > 1) {
```

```
 6.          int m = 0;
 7.
 8.          while (n > 0) {
 9.            int d = n % 10;
10.            m += d * d;
11.            n /= 10;
12.          }
13.
14.          if (set.contains(m))
15.            return false;
16.
17.          set.add(m);
18.
19.          n = m;
20.        }
21.
22.      return true;
23.    }
24.  }
```

Integer Break

- Integer Break

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).

Note: You may assume that n is not less than 2 and not larger than 58.

Hint:

1. There is a simple O(n) solution to this problem.
2. You may check the breaking results of n ranging from 7 to 10 to discover the regularities.

Largest Divisible Subset

- Largest Divisible Subset

Largest Divisible Subset

Given a set of **distinct** positive integers, find the largest subset such that every pair (Si, Sj) of elements in this subset satisfies: Si % Sj = 0 or Sj % Si = 0.

If there are multiple solutions, return any subset is fine.

Example 1:

```
1.  nums: [1,2,3]
2.
3.  Result: [1,2] (of course, [1,3] will also be ok)
```

Example 2:

```
1.  nums: [1,2,4,8]
2.
3.  Result: [1,2,4,8]
```

Line Reflection

Line Reflection

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

Example 1:

Given points = `[[1,1],[-1,1]]` , return `true` .

Example 2:

Given points = `[[1,1],[-1,-1]]` , return `false` .

Follow up:

Could you do better than O(n^2)?

Hint:

- Find the smallest and largest x-value for all points.
- If there is a line then it should be at y = (minX + maxX) / 2.
- For each point, make sure that it has a reflected point in the opposite side.

Max Points on a Line

Max Points on a Line

Given n points on a 2D plane, find the maximum number of points that lie on the same straight line.

Solution:

```java
/**
 * Definition for a point.
 * class Point {
 *     int x;
 *     int y;
 *     Point() { x = 0; y = 0; }
 *     Point(int a, int b) { x = a; y = b; }
 * }
 */
public class Solution {
  public int maxPoints(Point[] points) {
    if (points == null || points.length == 0)
      return 0;

    int n = points.length, max = 1;

    Map<String, Set<Point>> map = new HashMap<String, Set<Point>>();

    // sort the points by x to avoid -0.0 issue!
    Collections.sort(Arrays.asList(points), new Comparator<Point>() {
        public int compare(Point a, Point b) { return a.x - b.x; }
    });
```

```
24.        for (int i = 0; i < n; i++) {
25.          for (int j = i + 1; j < n; j++) {
26.            Point p1 = points[i];
27.            Point p2 = points[j];
28.
29.            StringBuilder sb = new StringBuilder();
30.
31.            if (p1.x == p2.x) {
32.                sb.append("inf").append(p1.x);
33.            } else {
34.                // y = k * x + d
35.                double k = (double)(p1.y - p2.y) / (p1.x - p2.x);
36.                double d = p1.y - k * p1.x;
37.                sb.append("k").append(k).append("d").append(d);
38.            }
39.
40.            String key = sb.toString();
41.            Set<Point> set = map.containsKey(key) ? map.get(key) : new
     HashSet<Point>();
42.
43.            set.add(p1);
44.            set.add(p2);
45.
46.            map.put(key, set);
47.            max = Math.max(max, set.size());
48.          }
49.        }
50.
51.      return max;
52.    }
53. }
```

Missing Number

- Missing Number

Missing Number

Given an array containing n distinct numbers taken from `0, 1, 2, ..., n` , find the one that is missing from the array.

For example,
Given nums = `[0, 1, 3]` return `2` .

Note:

Your algorithm should run in linear runtime complexity. Could you implement it using only constant extra space complexity?

Solution:

```java
 1.  public class Solution {
 2.    public int missingNumber(int[] nums) {
 3.      int sum = 0, n = nums.length;
 4.
 5.      for (int i = 0; i < n; i++)
 6.        sum += nums[i];
 7.
 8.      return n * (1 + n) / 2 - sum;
 9.    }
10.  }
```

Number of Digit One

- Number of Digit One

Number of Digit One

Given an integer n, count the total number of digit 1 appearing in all non-negative integers less than or equal to n.

For example:

Given n = 13,

Return 6, because digit 1 occurred in the following numbers: 1, 10, 11, 12, 13.

Hint:

Beware of overflow.

Solution:

```
public class Solution {
  public int countDigitOne(int n) {
    int count = 0;

    for (long k = 1; k <= n; k *= 10) {
      long r = n / k, m = n % k;
      count += (r + 8) / 10 * k + (r % 10 == 1 ? m + 1 : 0);
    }

    return count;
  }
}
```

Palindrome Number

- Palindrome Number

Palindrome Number

Determine whether an integer is a palindrome. Do this without extra space.

Some hints:

Could negative integers be palindromes? (ie, -1)

If you are thinking of converting the integer to string, note the restriction of using extra space.

You could also try reversing an integer. However, if you have solved the problem "Reverse Integer", you know that the reversed integer might overflow. How would you handle such case?

There is a more generic way of solving this problem.

Solution:

```java
public class Solution {
  public boolean isPalindrome(int x) {
    if (x < 0)
      return false;

    int s = 0, y = x;

    while (x > 0) {
      s = s * 10 + x % 10;
```

```
10.          x /= 10;
11.      }
12.
13.      return s == y;
14.    }
15.  }
```

Perfect Squares

- Perfect Squares

Perfect Squares

Given a positive integer n, find the least number of perfect square numbers (for example, `1, 4, 9, 16, ...`) which sum to n.

For example, given n = `12` , return `3` because `12 = 4 + 4 + 4` ; given n = `13` , return `2` because `13 = 4 + 9` .

Solution:

```
public class Solution {
  public int numSquares(int n) {
    // dp(i) represents the least number of PS which sum to i
    int[] dp = new int[n + 1];

    Arrays.fill(dp, Integer.MAX_VALUE);
    dp[0] = 0;

    for (int i = 0; i <= n; i++) {
      for (int j = 1; i + j * j <= n; j++) {
        dp[i + j * j] = Math.min(dp[i + j * j], dp[i] + 1);
      }
    }

    return dp[n];
  }
}
```

Permutation Sequence

- Permutation Sequence

Permutation Sequence

The set `[1,2,3,...,n]` contains a total of n! unique permutations.

By listing and labeling all of the permutations in order,
We get the following sequence (ie, for n = 3):

1. `"123"`
2. `"132"`
3. `"213"`
4. `"231"`
5. `"312"`
6. `"321"`
7. Given n and k, return the k-th permutation sequence.

Note: Given n will be between 1 and 9 inclusive.

Solution:

```
public class Solution {
  public String getPermutation(int n, int k) {
    // step 1. prepare the number sequence
    List<Integer> nums = new ArrayList<Integer>();
    for (int i = 1; i <= n; i++) {
      nums.add(i);
    }
```

```
 8.
 9.      // step 2. calculate (n-1)!
10.      int[] f = new int[n];
11.      f[0] = 1;
12.      for (int i = 1; i < n; i++) {
13.        f[i] = f[i - 1] * i;
14.      }
15.
16.      // step 3. calculate each digit, total n digits
17.      StringBuilder sb = new StringBuilder();
18.
19.      k--;
20.      while (n > 0) {
21.        int p = k  / f[n - 1];
22.
23.        sb.append(nums.get(p));
24.        nums.remove(p);
25.
26.        k = k % f[n - 1];
27.
28.        n--;
29.      }
30.
31.      return sb.toString();
32.    }
33. }
```

Plus One

- Plus One

Plus One

Given a non-negative number represented as an array of digits, plus one to the number.

The digits are stored such that the most significant digit is at the head of the list.

Solution:

```java
public class Solution {
  public int[] plusOne(int[] digits) {
    int n = digits.length, i = n - 1;

    for (; i >= 0; i--) {
      if (digits[i] == 9) {
        digits[i] = 0;
      } else {
        digits[i]++;
        return digits;
      }
    }

    int[] res = new int[n + 1];
    res[0] = 1;

    return res;
  }
}
```

Power of Three

- Power of Three

Power of Three

Given an integer, write a function to determine if it is a power of three.

Follow up:

Could you do it without using any loop / recursion?

Power of Two

-

Power of Two

Given an integer, write a function to determine if it is a power of two.

Solution:

```java
public class Solution {
  public boolean isPowerOfTwo(int n) {
    return (n > 0 && (n & (n - 1)) == 0);
  }
}
```

Pow

- Pow

Pow

Implement pow(x, n).

Solution:

```java
public class Solution {
  public double myPow(double x, int n) {
    if (x == 0) return 0;
    if (n == 0) return 1;

    double pow = myPow(x, Math.abs(n) / 2);
    double res = pow * pow;

    if ((n & 1) != 0) res *= x;

    return n < 0 ? 1 / res : res;
  }
}
```

Rectangle Area

- Rectangle Area

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 as shown in the figure.
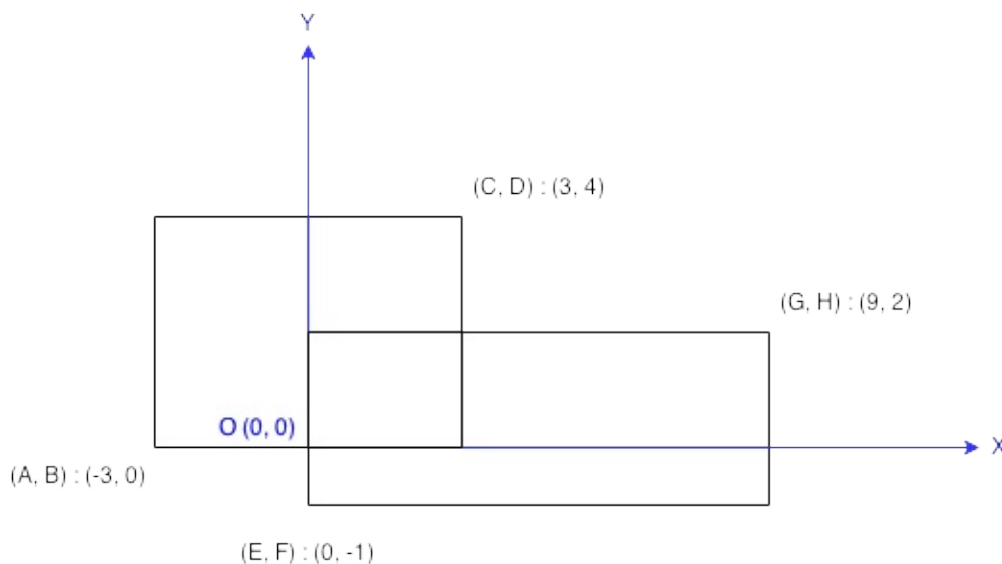
Assume that the total area is never beyond the maximum possible value of int.

Solution:

```
1.  public class Solution {
2.    public int computeArea(int A, int B, int C, int D, int E, int F, int G, int H) {
3.      long total = (C - A) * (D - B) + (G - E) * (H - F);
4.
```

```
 5.        long w = (long)Math.min(C, G) - (long)Math.max(A, E);
 6.        long h = (long)Math.min(D, H) - (long)Math.max(B, F);
 7.
 8.        long common = (w < 0 || h < 0) ? 0 : w * h;
 9.
10.        return (int)(total - common);
11.    }
12. }
```

Reverse Integer

-

Reverse Integer

Reverse digits of an integer.

Example1: x = `123` , return `321`

Example2: x = `-123` , return `-321`

Have you thought about this?

Here are some good questions to ask before coding. Bonus points for you if you have already thought through this!

If the integer's last digit is 0, what should the output be? ie, cases such as 10, 100.

Did you notice that the reversed integer might overflow? Assume the input is a 32-bit integer, then the reverse of 1000000003 overflows. How should you handle such cases?

For the purpose of this problem, assume that your function returns 0 when the reversed integer overflows.

Solution:

```
1.  public class Solution {
```

```java
 2.    public int reverse(int x) {
 3.      int y = Math.abs(x), s = 0;
 4.
 5.      while (y > 0) {
 6.        int m = y % 10;
 7.
 8.        if (x > 0) {
 9.          if (s > (Integer.MAX_VALUE - m) / 10) return 0;
10.          s = s * 10 + m;
11.        } else {
12.          if (s < (Integer.MIN_VALUE + m) / 10) return 0;
13.          s = s * 10 - m;
14.        }
15.
16.        y /= 10;
17.      }
18.
19.      return s;
20.    }
21.  }
```

Self Crossing

- Self Crossing

Self Crossing

You are given an array x of n positive numbers. You start at point (0,0) and moves x[0] metres to the north, then x[1] metres to the west, x[2] metres to the south, x[3] metres to the east and so on. In other words, after each move your direction changes counter-clockwise.
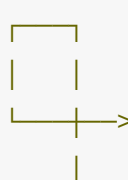
Write a one-pass algorithm with O(1) extra space to determine, if your path crosses itself, or not.

Example 1:

```
1.  Given x = [2, 1, 1, 2],
2.  ┌──┐
3.  │  │
4.  └──┼──>
5.     │
6.
7.  Return true (self crossing)
```
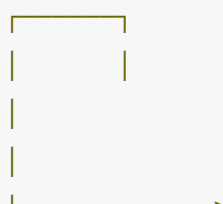
Example 2:

```
1.  Given x = [1, 2, 3, 4],
2.  ┌────┐
3.  │    │
4.  │
5.  │
6.  └──────────>
```

```
7.
8.  Return false (not self crossing)
```

Example 3:

```
1.  Given x = [1, 1, 1, 1],
2.  ┌───┐
3.  │   │
4.  └───┼>
5.
6.  Return true (self crossing)
```

Sqrt

- Sqrt

Sqrt

Implement `int sqrt(int x)` .

Compute and return the square root of x.

Solution:

```
public class Solution {
  public int mySqrt(int x) {
    int lo = 1, hi = x;

    while (lo < hi) {
      int mid = lo + (hi - lo) / 2;

      if (mid < x / mid) {
        lo = mid + 1;
      } else {
        hi = mid;
      }
    }

    return (lo == x / lo) ? lo : lo - 1;
  }
}
```

Strobogrammatic Number

- Strobogrammatic Number

Strobogrammatic Number

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to determine if a number is strobogrammatic. The number is represented as a string.

For example, the numbers "69", "88", and "818" are all strobogrammatic.

Solution:

```java
public class Solution {
  public boolean isStrobogrammatic(String num) {
    if (num == null)
      return false;

    return helper(num, 0, num.length() - 1);
  }

  boolean helper(String s, int lo, int hi) {
    if (lo > hi)
      return true;

    char c1 = s.charAt(lo);
    char c2 = s.charAt(hi);

```

```
16.        int mul = (c1 - '0') * (c2 - '0');
17.
18.        if (mul == 1 || mul == 54 || mul == 64 || (mul == 0 && c1 ==
    c2))
19.            return helper(s, lo + 1, hi - 1);
20.
21.        return false;
22.    }
23. }
```

Super Ugly Number

Super Ugly Number

Write a program to find the nth super ugly number.

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

Note:

(1) `1` is a super ugly number for any given `primes` .

(2) The given numbers in `primes` are in ascending order.

(3) 0 < `k` ≤ 100, 0 < `n` ≤ 10^6, 0 < `primes[i]` < 1000.

Ugly Number II

- Ugly Number II

Ugly Number II

Write a program to find the n-th ugly number.

Ugly numbers are positive numbers whose prime factors only include `2, 3, 5` . For example, `1, 2, 3, 4, 5, 6, 8, 9, 10, 12` is the sequence of the first `10` ugly numbers.

Note that `1` is typically treated as an ugly number.

Hint:

1. The naive approach is to call isUgly for every number until you reach the nth one. Most numbers are not ugly. Try to focus your effort on generating only the ugly ones.
2. An ugly number must be multiplied by either 2, 3, or 5 from a smaller ugly number.
3. The key is how to maintain the order of the ugly numbers. Try a similar approach of merging from three sorted lists: L1, L2, and L3.
4. Assume you have Uk, the kth ugly number. Then Uk+1 must be Min(L1 *2, L2* 3, L3 * 5).

Solution:

```java
public class Solution {
  public int nthUglyNumber(int n) {
    // 1x2, 2x2, 3x2, 4x2, 5x2 ...
    // 1x3, 2x3, 3x3, 4x3, 5x3 ...
    // 1x5, 2x5, 3x5, 4x5, 5x5 ...

    int[] ugly = new int[n + 1];
    ugly[1] = 1;

    int next2 = 2, next3 = 3, next5 = 5;
    int i2 = 1, i3 = 1, i5 = 1;

    for (int i = 2; i <= n; i++) {
      int min = Math.min(next2, Math.min(next3, next5));

      ugly[i] = min;

      if (min == next2)
        next2 = ugly[++i2] * 2;

      if (min == next3)
        next3 = ugly[++i3] * 3;

      if (min == next5)
        next5 = ugly[++i5] * 5;
    }

    return ugly[n];
  }
}
```

Ugly Number

- Ugly Number

Ugly Number

Write a program to check whether a given number is an ugly number.

Ugly numbers are positive numbers whose prime factors only include 2, 3, 5. For example, 6, 8 are ugly while 14 is not ugly since it includes another prime factor 7.

Note that 1 is typically treated as an ugly number.

Solution:

```
1.  public class Solution {
2.    public boolean isUgly(int num) {
3.      if (num == 0)
4.        return false;
5.
6.      while (num != 1) {
7.        if (num % 2 == 0) {
8.          num /= 2;
9.        } else if (num % 3 == 0) {
10.         num /= 3;
11.       } else if (num % 5 == 0) {
12.         num /= 5;
13.       } else {
14.         return false;
15.       }
16.     }
17.
```

```
18.        return true;
19.    }
20. }
```

Valid Perfect Square

- Valid Perfect Square

Valid Perfect Square

Given a positive integer num, write a function which returns True if num is a perfect square else False.

Note: Do not use any built-in library function such as sqrt.

Example 1:

```
1.   Input: 16
2.   Returns: True
```

Example 2:

```
1.   Input: 14
2.   Returns: False
```

Water and Jug Problem

Water and Jug Problem

You are given two jugs with capacities x and y litres. There is an infinite amount of water supply available. You need to determine whether it is possible to measure exactly z litres using these two jugs.

If z liters of water is measurable, you must have z liters of water contained within one or both buckets by the end.

Operations allowed:

- Fill any of the jugs completely with water.
- Empty any of the jugs.
- Pour water from one jug into another till the other jug is completely full or the first jug itself is empty.

Example 1: (From the famous "Die Hard" example)

```
1.  Input: x = 3, y = 5, z = 4
2.  Output: True
```

Example 2:

```
1.  Input: x = 2, y = 6, z = 5
```

```
2.  Output: False
```

Memoization

- Memoization

Memoization

Queue

- Queue

Queue

Design Snake Game

Design Snake Game

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

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

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

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

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

Example:

```
1.  Given width = 3, height = 2, and food = [[1,2],[0,1]].
2.
3.  Snake snake = new Snake(width, height, food);
4.
5.  Initially the snake appears at position (0,0) and the food at
    (1,2).
```

```
  6.
  7.  |S| | |
  8.  | | |F|
  9.
 10.  snake.move("R"); -> Returns 0
 11.
 12.  | |S| |
 13.  | | |F|
 14.
 15.  snake.move("D"); -> Returns 0
 16.
 17.  | | | |
 18.  | |S|F|
 19.
 20.  snake.move("R"); -> Returns 1 (Snake eats the first food and right
      after that, the second food appears at (0,1) )
 21.
 22.  | |F| |
 23.  | |S|S|
 24.
 25.  snake.move("U"); -> Returns 1
 26.
 27.  | |F|S|
 28.  | | |S|
 29.
 30.  snake.move("L"); -> Returns 2 (Snake eats the second food)
 31.
 32.  | |S|S|
 33.  | | |S|
 34.
 35.  snake.move("U"); -> Returns -1 (Game over because snake collides
      with border)
```

Max Sum of Rectangle No Larger Than K

- Max Sum of Rectangle No Larger Than K

Max Sum of Rectangle No Larger Than K

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

Example:

```
1.  Given matrix = [
2.    [1,  0, 1],
3.    [0, -2, 3]
4.  ]
5.  k = 2
```

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

Note:

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

Moving Average from Data Stream

- Moving Average from Data Stream

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.

For example,

```
1.  MovingAverage m = new MovingAverage(3);
2.  m.next(1) = 1
3.  m.next(10) = (1 + 10) / 2
4.  m.next(3) = (1 + 10 + 3) / 3
5.  m.next(5) = (10 + 3 + 5) / 3
6.  Show Company Tags
7.  Show Tags
```

Recursion

- Recursion

Recursion

Strobogrammatic Number II

- Strobogrammatic Number II

Strobogrammatic Number II

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Find all strobogrammatic numbers that are of length = n.

For example,

Given n = 2, return `["11","69","88","96"]` .

Hint:

Try to use recursion and notice that it should recurse with n - 2 instead of n - 1.

Solution:

```
1.  public class Solution {
2.    public List<String> findStrobogrammatic(int n) {
3.      return helper(n, n);
4.    }
5.
6.    List<String> helper(int n, int m) {
7.      if (n == 0) return new ArrayList<String>(Arrays.asList(""));
8.      if (n == 1) return new ArrayList<String>(Arrays.asList("0",
     "1", "8"));
9.
10.     List<String> list = helper(n - 2, m);
```

```
11.
12.       List<String> res = new ArrayList<String>();
13.
14.       for (int i = 0; i < list.size(); i++) {
15.         String s = list.get(i);
16.
17.         if (n != m) res.add("0" + s + "0");
18.
19.         res.add("1" + s + "1");
20.         res.add("6" + s + "9");
21.         res.add("8" + s + "8");
22.         res.add("9" + s + "6");
23.       }
24.
25.       return res;
26.     }
27. }
```

Strobogrammatic Number III

Strobogrammatic Number III

A strobogrammatic number is a number that looks the same when rotated 180 degrees (looked at upside down).

Write a function to count the total strobogrammatic numbers that exist in the range of low <= num <= high.

For example,
Given low = "50", high = "100", return 3. Because 69, 88, and 96 are three strobogrammatic numbers.

Note:

Because the range might be a large number, the low and high numbers are represented as string.

Solution:

```
1.  public class Solution {
2.    int count = 0;
3.    String[][] pairs = {{"0", "0"}, {"1", "1"}, {"6", "9"}, {"8",
    "8"}, {"9", "6"}};
4.
5.    public int strobogrammaticInRange(String low, String high) {
6.      Map<Integer, List<String>> map = new HashMap<Integer,
    List<String>>();
7.      map.put(0, new ArrayList<String>(Arrays.asList("")));
```

```
 8.        map.put(1, new ArrayList<String>(Arrays.asList("0", "1",
    "8")));
 9.
10.        for (int i = low.length(); i <= high.length(); i++)
11.          helper(i, map, low, high);
12.
13.        return count;
14.      }
15.
16.      List<String> helper(int n, Map<Integer, List<String>> map, String
    low, String high) {
17.        List<String> res = new ArrayList<String>();
18.
19.        if (map.containsKey(n)) {
20.          res = map.get(n);
21.        } else {
22.          List<String> list = map.containsKey(n - 2) ? map.get(n - 2) :
    helper(n - 2, map, low, high);
23.
24.          for (int i = 0; i < list.size(); i++) {
25.            String s = list.get(i);
26.
27.            for (int j = 0; j < pairs.length; j++) {
28.              String v = pairs[j][0] + s + pairs[j][1];
29.
30.              if (v.length() == high.length() && v.compareTo(high) > 0)
31.                break;
32.
33.              res.add(v);
34.            }
35.          }
36.
37.          map.put(n, res);
38.        }
39.
40.        if (n >= low.length()) {
41.          count += res.size();
42.
```

```
43.          for (String s : res) {
44.            if ((s.length() > 1 && s.charAt(0) == '0') || (s.length()
     == low.length() && s.compareTo(low) < 0) || (s.length() ==
     high.length() && s.compareTo(high) > 0))
45.                count--;
46.        }
47.      }
48.
49.      return res;
50.    }
51.  }
```

Segment Tree

- Segment Tree

Segment Tree

Count of Smaller Numbers After Self

- Count of Smaller Numbers After Self

Count of Smaller Numbers After Self

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

Example:

```
1.  Given nums = [5, 2, 6, 1]
2.
3.  To the right of 5 there are 2 smaller elements (2 and 1).
4.  To the right of 2 there is only 1 smaller element (1).
5.  To the right of 6 there is 1 smaller element (1).
6.  To the right of 1 there is 0 smaller element.
```

Return the array [2, 1, 1, 0] .

Solution:

```
1.  public class Solution {
2.    private void add(int[] bit, int i, int val) {
3.      for (; i < bit.length; i += i & -i) bit[i] += val;
4.    }
5.
6.    private int query(int[] bit, int i) {
7.      int ans = 0;
8.      for (; i > 0; i -= i & -i) ans += bit[i];
9.      return ans;
10.   }
11.
```

```
12.    public List<Integer> countSmaller(int[] nums) {
13.      int[] tmp = nums.clone();
14.      Arrays.sort(tmp);
15.      for (int i = 0; i < nums.length; i++) nums[i] =
    Arrays.binarySearch(tmp, nums[i]) + 1;
16.      int[] bit = new int[nums.length + 1];
17.      Integer[] ans = new Integer[nums.length];
18.      for (int i = nums.length - 1; i >= 0; i--) {
19.        ans[i] = query(bit, nums[i] - 1);
20.        add(bit, nums[i], 1);
21.      }
22.      return Arrays.asList(ans);
23.    }
24. }
```

Range Sum Query 2D - Mutable

Range Sum Query 2D - Mutable

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).

The above rectangle (with the red border) is defined by (row1, col1) = (2, 1) and (row2, col2) = (4, 3), which contains sum = 8.

Example:

```
1.  Given matrix = [
2.    [3, 0, 1, 4, 2],
3.    [5, 6, 3, 2, 1],
4.    [1, 2, 0, 1, 5],
5.    [4, 1, 0, 1, 7],
6.    [1, 0, 3, 0, 5]
7.  ]
8.
9.  sumRegion(2, 1, 4, 3) -> 8
10. update(3, 2, 2)
```

```
11.  sumRegion(2, 1, 4, 3) -> 10
```

Solution:

```java
1.  public class NumMatrix {
2.
3.    int m, n;
4.    int[][] arr;    // stores matrix[][]
5.    int[][] BITree; // 2-D binary indexed tree
6.
7.    public NumMatrix(int[][] matrix) {
8.      if (matrix.length == 0 || matrix[0].length == 0) {
9.          return;
10.     }
11.
12.     m = matrix.length;
13.     n = matrix[0].length;
14.
15.     arr = new int[m][n];
16.     BITree = new int[m + 1][n + 1];
17.
18.     for (int i = 0; i < m; i++) {
19.       for (int j = 0; j < n; j++) {
20.         update(i, j, matrix[i][j]); // init BITree[][]
21.         arr[i][j] = matrix[i][j];   // init arr[][]
22.       }
23.     }
24.   }
25.
26.   public void update(int i, int j, int val) {
27.     int diff = val - arr[i][j];
28.     arr[i][j] = val;
29.
30.     i++; j++;
31.     while (i <= m) {
32.       int k = j;
33.       while (k <= n) {
34.         BITree[i][k] += diff;
```

```
35.          k += k & (-k);
36.        }
37.        i += i & (-i);
38.      }
39.    }
40.
41.    int getSum(int i, int j) {
42.      int sum = 0;
43.
44.      i++; j++;
45.      while (i > 0) {
46.        int k = j;
47.        while (k > 0) {
48.          sum += BITree[i][k];
49.          k -= k & (-k);
50.        }
51.        i -= i & (-i);
52.      }
53.      return sum;
54.    }
55.
56.    public int sumRegion(int i1, int j1, int i2, int j2) {
57.      return getSum(i2, j2) - getSum(i1-1, j2) - getSum(i2, j1-1) +
    getSum(i1-1, j1-1);
58.    }
59. }
60.
61.
62. // Your NumMatrix object will be instantiated and called as such:
63. // NumMatrix numMatrix = new NumMatrix(matrix);
64. // numMatrix.sumRegion(0, 1, 2, 3);
65. // numMatrix.update(1, 1, 10);
66. // numMatrix.sumRegion(1, 2, 3, 4);
```

Range Sum Query - Mutable

Range Sum Query - Mutable

Given an integer array nums, find the sum of the elements between indices i and j (i ≤ j), inclusive.

The update(i, val) function modifies nums by updating the element at index i to val.

Example:

```
1.  Given nums = [1, 3, 5]
2.
3.  sumRange(0, 2) -> 9
4.  update(1, 2)
5.  sumRange(0, 2) -> 8
```

Note:

1. The array is only modifiable by the update function.
2. You may assume the number of calls to update and sumRange function is distributed evenly.

Solution:

```
1.  public class NumArray {
2.    int[] arr;    // stores nums[]
3.    int[] BITree; // binary indexed tree
4.
5.    public NumArray(int[] nums) {
```

```
 6.        int n = nums.length;
 7.        arr = new int[n];
 8.        BITree = new int[n + 1];
 9.
10.        for (int i = 0; i < n; i++) {
11.          update(i, nums[i]); // init BITree[]
12.          arr[i] = nums[i];   // init arr[]
13.        }
14.    }
15.
16.    void update(int i, int val) {
17.      int diff = val - arr[i]; // get the diff
18.      arr[i] = val;            // update arr[]
19.
20.      i++;
21.      while (i <= arr.length) {
22.        BITree[i] += diff; // update BITree[]
23.        i += i & (-i);     // update index to that of parent
24.      }
25.    }
26.
27.    int getSum(int i) {
28.      int sum = 0;
29.
30.      i++;
31.      while (i > 0) {
32.        sum += BITree[i]; // accumulate the sum
33.        i -= i & (-i);    // move index to parent node
34.      }
35.      return sum;
36.    }
37.
38.    public int sumRange(int i, int j) {
39.      return getSum(j) - getSum(i - 1);
40.    }
41.  }
42.
43.  // Your NumArray object will be instantiated and called as such:
```

```
44.  // NumArray numArray = new NumArray(nums);
45.  // numArray.sumRange(0, 1);
46.  // numArray.update(1, 10);
47.  // numArray.sumRange(1, 2);
```

The Skyline Problem

The Skyline Problem

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are **given the locations and height of all the buildings** as shown on a cityscape photo (Figure A), write a program to output the skyline formed by these buildings collectively (Figure B).
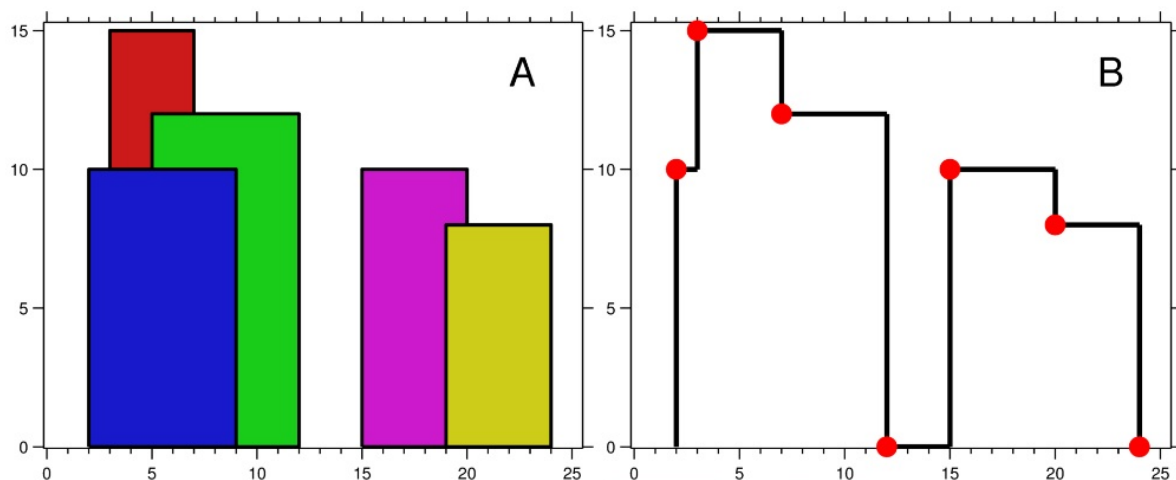
The geometric information of each building is represented by a triplet of integers `[Li, Ri, Hi]` , where `Li` and `Ri` are the x coordinates of the left and right edge of the ith building, respectively, and Hi is its height. It is guaranteed that `0 ≤ Li, Ri ≤ INT_MAX, 0 < Hi ≤ INT_MAX, and Ri - Li > 0` . You may assume all buildings are perfect rectangles

grounded on an absolutely flat surface at height 0.

For instance, the dimensions of all buildings in Figure A are recorded as: `[ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ]` .

The output is a list of **"key points"** (red dots in Figure B) in the format of `[ [x1,y1], [x2, y2], [x3, y3], ... ]` that uniquely defines a skyline. **A key point is the left endpoint of a horizontal line segment**. Note that the last key point, where the rightmost building ends, is merely used to mark the termination of the skyline, and always has zero height. Also, the ground in between any two adjacent buildings should be considered part of the skyline contour.

For instance, the skyline in Figure B should be represented as: `[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ]` .

Notes:

- The number of buildings in any input list is guaranteed to be in the range `[0, 10000]` .
- The input list is already sorted in ascending order by the left x position `Li` .
- The output list must be sorted by the x position.
- There must be no consecutive horizontal lines of equal height in the output skyline. For

instance, `[...[2 3], [4 5], [7 5], [11 5], [12 7]...]` is not acceptable; the three lines of height 5 should be merged into one in the final output as such: [...[2 3], [4 5], [12 7], ...]

Solution:

```java
public class Solution {
  public List<int[]> getSkyline(int[][] buildings) {
    return getSkyline(buildings, 0, buildings.length - 1);
  }

  List<int[]> getSkyline(int[][] b, int lo, int hi) {
    List<int[]> res = new ArrayList<>();

    if (lo > hi) return res;

    if (lo == hi) {
      res.add(new int[]{b[lo][0], b[lo][2]});
      res.add(new int[]{b[lo][1], 0});
      return res;
    }

    int mid = lo + (hi - lo) / 2;

    return merge(getSkyline(b, lo, mid), getSkyline(b, mid + 1, hi));
  }

  List<int[]> merge(List<int[]> left, List<int[]> right) {
    List<int[]> res = new ArrayList<>();

    int i = 0, j = 0, h1 = 0, h2 = 0, n = left.size(), m = right.size();

    while (i < n && j < m) {
      int[] l = left.get(i);
```

```
29.        int[] r = right.get(j);
30.
31.        if (l[0] < r[0]) {
32.          h1 = l[1];
33.          append(res, new int[]{l[0], Math.max(h1, h2)});
34.          i++;
35.        } else if (r[0] < l[0]) {
36.          h2 = r[1];
37.          append(res, new int[]{r[0], Math.max(h1, h2)});
38.          j++;
39.        } else {
40.          h1 = l[1];
41.          h2 = r[1];
42.          append(res, new int[]{l[0], Math.max(h1, h2)});
43.          i++;
44.        }
45.      }
46.
47.      while (i < n) {
48.        append(res, left.get(i++));
49.      }
50.
51.      while (j < m) {
52.        append(res, right.get(j++));
53.      }
54.
55.      return res;
56.    }
57.
58.    void append(List<int[]> res, int[] strip) {
59.      int n = res.size();
60.
61.      if (n > 0 && res.get(n - 1)[1] == strip[1])
62.        return;
63.
64.      if (n > 0 && res.get(n - 1)[0] == strip[0]) {
65.        res.get(n - 1)[1] = Math.max(res.get(n - 1)[1], strip[1]);
66.        return;
```

```
67.        }
68.
69.     res.add(strip);
70.   }
71. }
```

Sort

- Sort

Sort

Best Meeting Point

- Best Meeting Point

Best Meeting Point

A group of two or more people wants to meet and minimize the total travel distance. You are given a 2D grid of values 0 or 1, where each 1 marks the home of someone in the group. The distance is calculated using Manhattan Distance, where `distance(p1, p2) = |p2.x - p1.x| + |p2.y - p1.y|` .

For example, given three people living at `(0,0)` , `(0,4)` , and `(2,2)` :

```
1. 1 - 0 - 0 - 0 - 1
2. |   |   |   |   |
3. 0 - 0 - 0 - 0 - 0
4. |   |   |   |   |
5. 0 - 0 - 1 - 0 - 0
```

The point `(0,2)` is an ideal meeting point, as the total travel distance of `2+2+2=6` is minimal. So return `6` .

Hint:

Try to solve it in one dimension first. How can this solution apply to the two dimension case?

Solution:

```java
import java.util.Random;

public class Solution {
  public int minTotalDistance(int[][] grid) {
    List<Integer> x = new ArrayList<>();
    List<Integer> y = new ArrayList<>();

    for (int i = 0; i < grid.length; i++) {
      for (int j = 0; j < grid[0].length; j++) {
        if (grid[i][j] == 1) {
          x.add(i); y.add(j);
        }
      }
    }

    // get median of x[] and y[] using quick select
    int mx = x.get(quickSelect(x, 0, x.size() - 1, x.size() / 2 +
1));
    int my = y.get(quickSelect(y, 0, y.size() - 1, y.size() / 2 +
1));

    // calculate the total Manhattan distance
    int total = 0;
    for (int i = 0; i < x.size(); i++) {
      total += Math.abs(x.get(i) - mx) + Math.abs(y.get(i) - my);
    }
    return total;
  }

  // return the index of the kth smallest number
  // avg. O(n) time complexity
  int quickSelect(List<Integer> a, int lo, int hi, int k) {
    // use quick sort's idea
    // randomly pick a pivot and put it to a[hi]
    // we need to do this, otherwise quick sort is slow!
    Random rand = new Random();
    int p = lo + rand.nextInt(hi - lo + 1);
    Collections.swap(a, p, hi);
```

```
37.
38.      // put nums that are <= pivot to the left
39.      // put nums that are  > pivot to the right
40.      int i = lo, j = hi, pivot = a.get(hi);
41.      while (i < j) {
42.        if (a.get(i++) > pivot) Collections.swap(a, --i, --j);
43.      }
44.      Collections.swap(a, i, hi);
45.
46.      // count the nums that are <= pivot from lo
47.      int m = i - lo + 1;
48.
49.      // pivot is the one!
50.      if (m == k)     return i;
51.      // pivot is too big, so it must be on the left
52.      else if (m > k) return quickSelect(a, lo, i - 1, k);
53.      // pivot is too small, so it must be on the right
54.      else            return quickSelect(a, i + 1, hi, k - m);
55.    }
56.  }
```

H-Index

- H-Index

H-Index

Given an array of citations (each citation is a non-negative integer) of a researcher, write a function to compute the researcher's h-index.

According to the definition of h-index on Wikipedia: "A scientist has index h if h of his/her N papers have **at least** h citations each, and the other N − h papers have **no more than** h citations each."

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

Note: If there are several possible values for `h`, the maximum one is taken as the h-index.

Hint:

- An easy approach is to sort the array first.
- What are the possible values of h-index?
- A faster approach is to use extra space.

Solution:

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

    int n = citations.length, max = 0;

    for (int i = 0; i < n; i++)
      max = Math.max(max, Math.min(citations[i], n - i));

    return max;
  }
}
```

Insert Interval

- Insert Interval

Insert Interval

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

You may assume that the intervals were initially sorted according to their start times.

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]` .

Solution:

```
1.  /**
2.   * Definition for an interval.
3.   * public class Interval {
4.   *     int start;
5.   *     int end;
6.   *     Interval() { start = 0; end = 0; }
```

```
 7.  *     Interval(int s, int e) { start = s; end = e; }
 8.  * }
 9.  */
10. public class Solution {
11.   public List<Interval> insert(List<Interval> intervals, Interval
    newInterval) {
12.     List<Interval> res = new ArrayList<>();
13.
14.     int i = 0, n = intervals.size();
15.
16.     while (i < n && intervals.get(i).end < newInterval.start) {
17.       res.add(intervals.get(i++));
18.     }
19.
20.     while (i < n && intervals.get(i).start <= newInterval.end) {
21.       newInterval = new Interval(
22.         Math.min(newInterval.start, intervals.get(i).start),
23.         Math.max(newInterval.end, intervals.get(i).end)
24.       );
25.       i++;
26.     }
27.
28.     res.add(newInterval);
29.
30.     while (i < n) {
31.       res.add(intervals.get(i++));
32.     }
33.
34.     return res;
35.   }
36. }
```

Insertion Sort List

- Insertion Sort List

Insertion Sort List

Sort a linked list using insertion sort.

Solution:

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
  public ListNode insertionSortList(ListNode head) {
    ListNode curr = head, next = null;
    ListNode l = new ListNode(0);

    while (curr != null) {
      next = curr.next;

      ListNode p = l;
      while (p.next != null && p.next.val < curr.val) {
          p = p.next;
      }

      curr.next = p.next;
      p.next = curr;
      curr = next;
    }

```

```
27.        return l.next;
28.    }
29. }
```

Largest Number

- Largest Number

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.

Solution:

```
1.  public class Solution {
2.    public String largestNumber(int[] num) {
3.      int n = num.length;
4.
5.      Integer[] A = new Integer[n];
6.      for (int i = 0; i < n; i++) {
7.        A[i] = Integer.valueOf(num[i]);
8.      }
9.
10.     // sort the array by string
11.     Arrays.sort(A, new Comparator<Integer>() {
12.       @Override
13.       public int compare(Integer a, Integer b) {
14.         String stra = String.valueOf(a);
15.         String strb = String.valueOf(b);
16.
17.         return (strb + stra).compareTo(stra + strb);
18.       }
```

```
19.        });
20.
21.        StringBuilder sb = new StringBuilder();
22.
23.        for (int i = 0; i < n; i++) {
24.            if (A[i] == 0 && sb.length() == 0) {
25.                continue;
26.            }
27.
28.            sb.append(String.valueOf(A[i]));
29.        }
30.
31.        if (sb.length() == 0) {
32.            return "0";
33.        }
34.
35.        return sb.toString();
36.    }
37. }
```

Maximum Gap

- Maximum Gap

Maximum Gap

Given an unsorted array, find the maximum difference between the successive elements in its sorted form.

Try to solve it in linear time/space.

Return 0 if the array contains less than 2 elements.

You may assume all elements in the array are non-negative integers and fit in the 32-bit signed integer range.

Solution:

```java
1.  public class Solution {
2.    public int maximumGap(int[] nums) {
3.        if (nums == null || nums.length < 2) {
4.          return 0;
5.        }
6.
7.        int n = nums.length;
8.
9.        // m is the maximal number in nums
10.       int m = nums[0];
11.       for (int i = 1; i < n; i++) {
12.         m = Math.max(m, nums[i]);
13.       }
14.
15.       int exp = 1; // 1, 10, 100, 1000 ...
16.       int R = 10; // 10 digits
```

```
17.
18.        int[] aux = new int[n];
19.
20.        while (m / exp > 0) { // Go through all digits from LSB to
     MSB
21.          int[] count = new int[R];
22.
23.          for (int i = 0; i < n; i++) {
24.            count[(nums[i] / exp) % 10]++;
25.          }
26.
27.          for (int i = 1; i < R; i++) {
28.            count[i] += count[i - 1];
29.          }
30.
31.          for (int i = n - 1; i >= 0; i--) {
32.            aux[--count[(nums[i] / exp) % 10]] = nums[i];
33.          }
34.
35.          for (int i = 0; i < n; i++) {
36.              nums[i] = aux[i];
37.          }
38.
39.          exp *= 10;
40.        }
41.
42.        int max = 0;
43.        for (int i = 1; i < aux.length; i++) {
44.          max = Math.max(max, aux[i] - aux[i - 1]);
45.        }
46.
47.        return max;
48.    }
49. }
```

Meeting Rooms

- Meeting Rooms

Meeting Rooms

Given an array of meeting time intervals consisting of start and end times `[[s1,e1],[s2,e2],...]` (si < ei), determine if a person could attend all meetings.

For example,

Given `[[0, 30],[5, 10],[15, 20]]` ,

return `false` .

Solution:

```
 1.  /**
 2.   * Definition for an interval.
 3.   * public class Interval {
 4.   *     int start;
 5.   *     int end;
 6.   *     Interval() { start = 0; end = 0; }
 7.   *     Interval(int s, int e) { start = s; end = e; }
 8.   * }
 9.   */
10.  public class Solution {
11.    public boolean canAttendMeetings(Interval[] intervals) {
12.      // Sort the intervals by start time
13.      Arrays.sort(intervals, new Comparator<Interval>() {
14.        public int compare(Interval a, Interval b) { return a.start - b.start; }
15.      });
16.
17.      for (int i = 1; i < intervals.length; i++)
```

```
18.         if (intervals[i].start < intervals[i - 1].end)
19.           // found a conflict
20.           return false;
21.
22.       return true;
23.   }
24. }
```

Meeting Rooms II

- Meeting Rooms II

Meeting Rooms II

Given an array of meeting time intervals consisting of start and end times `[[s1,e1],[s2,e2],...]` (si < ei), find the minimum number of conference rooms required.

For example,

Given `[[0, 30],[5, 10],[15, 20]]` ,

return `2` .

Solution:

```
1.  /**
2.   * Definition for an interval.
3.   * public class Interval {
4.   *     int start;
5.   *     int end;
6.   *     Interval() { start = 0; end = 0; }
7.   *     Interval(int s, int e) { start = s; end = e; }
8.   * }
9.   */
10. public class Solution {
11.   public int minMeetingRooms(Interval[] intervals) {
12.     if (intervals == null || intervals.length == 0)
13.       return 0;
14.
15.     // Sort the intervals by start time
16.     Arrays.sort(intervals, new Comparator<Interval>() {
```

```
17.        public int compare(Interval a, Interval b) { return a.start -
    b.start; }
18.      });
19.
20.      // Use a min heap to track the minimum end time of merged
    intervals
21.      PriorityQueue<Interval> heap = new PriorityQueue<Interval>
    (intervals.length, new Comparator<Interval>() {
22.        public int compare(Interval a, Interval b) { return a.end -
    b.end; }
23.      });
24.
25.      heap.offer(intervals[0]);
26.
27.      for (int i = 1; i < intervals.length; i++) {
28.        Interval interval = heap.poll();
29.
30.        if (intervals[i].start >= interval.end) {
31.          interval.end = intervals[i].end;
32.        } else {
33.          heap.offer(intervals[i]);
34.        }
35.
36.        heap.offer(interval);
37.      }
38.
39.      return heap.size();
40.    }
41.  }
```

Merge Intervals

- Merge Intervals

Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

For example,

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

Solution:

```
1.  /**
2.   * Definition for an interval.
3.   * public class Interval {
4.   *     int start;
5.   *     int end;
6.   *     Interval() { start = 0; end = 0; }
7.   *     Interval(int s, int e) { start = s; end = e; }
8.   * }
9.   */
10. public class Solution {
11.   public List<Interval> merge(List<Interval> intervals) {
12.     // sort
13.     Collections.sort(intervals, new Comparator<Interval>() {
14.       public int compare(Interval a, Interval b) {
15.         return a.start - b.start;
16.       }
17.     });
18.
19.     // merge
20.     Stack<Interval> stack = new Stack<Interval>();
```

```
21.
22.      for (int i = 0; i < intervals.size(); i++) {
23.        Interval curr = intervals.get(i);
24.
25.        if (stack.isEmpty() || curr.start > stack.peek().end) {
26.          stack.push(curr);
27.        } else {
28.          stack.peek().end = Math.max(curr.end, stack.peek().end);
29.        }
30.      }
31.
32.      // return
33.      List<Interval> res = new ArrayList<Interval>();
34.      while (!stack.isEmpty()) {
35.        res.add(stack.pop());
36.      }
37.
38.      return res;
39.    }
40.  }
```

Sort List

- Sort List

Sort List

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

Solution:

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

    // step 1. cut the list to two halves
    ListNode prev = null, slow = head, fast = head;

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

    prev.next = null;

    // step 2. sort each half
```

```
26.        ListNode l1 = sortList(head);
27.        ListNode l2 = sortList(slow);
28.
29.        // step 3. merge l1 and l2
30.        return merge(l1, l2);
31.    }
32.
33.    ListNode merge(ListNode l1, ListNode l2) {
34.        ListNode l = new ListNode(0), p = l;
35.
36.        while (l1 != null && l2 != null) {
37.            if (l1.val < l2.val) {
38.                p.next = l1;
39.                l1 = l1.next;
40.            } else {
41.                p.next = l2;
42.                l2 = l2.next;
43.            }
44.            p = p.next;
45.        }
46.
47.        if (l1 != null)
48.            p.next = l1;
49.
50.        if (l2 != null)
51.            p.next = l2;
52.
53.        return l.next;
54.    }
55. }
```

Valid Anagram

- Valid Anagram

Valid Anagram

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

For example,

s = `"anagram"` , t = `"nagaram"` , return `true` .

s = `"rat"` , t = `"car"` , return `false` .

Note:

You may assume the string contains only lowercase alphabets.

Follow up:

What if the inputs contain unicode characters? How would you adapt your solution to such case?

Solution:

```
1.  public class Solution {
2.    public boolean isAnagram(String s, String t) {
3.        if (s == null || t == null || s.length() != t.length())
4.          return false;
5.
6.        int[] arr = new int[128];
7.
8.        for (int i = 0; i < s.length(); i++) {
```

```
 9.          arr[(int)s.charAt(i)]++;
10.          arr[(int)t.charAt(i)]--;
11.        }
12.
13.        for (int i = 0; i < arr.length; i++) {
14.          if (arr[i] != 0)
15.            return false;
16.        }
17.
18.        return true;
19.    }
20.  }
```

Wiggle Sort

- Wiggle Sort

Wiggle Sort

Given an unsorted array `nums` , reorder it **in-place** such that `nums[0] <= nums[1] >= nums[2] <= nums[3]...` .

For example, given nums = `[3, 5, 2, 1, 6, 4]` , one possible answer is `[1, 6, 2, 5, 3, 4]` .

Solution:

```
1.  public class Solution {
2.    public void wiggleSort(int[] nums) {
3.      for (int i = 1; i < nums.length; i++) {
4.        if (i % 2 != 0 && nums[i] < nums[i - 1]) {
5.          swap(nums, i, i - 1);
6.        } else if (i % 2 == 0 && nums[i] > nums[i - 1]) {
7.          swap(nums, i, i - 1);
8.        }
9.      }
10.   }
11.
12.   void swap(int[] nums, int i, int j) {
13.     int tmp = nums[i];
14.     nums[i] = nums[j];
15.     nums[j] = tmp;
16.   }
17. }
```

Wiggle Sort II

- Wiggle Sort II

Wiggle Sort II

Given an unsorted array `nums` , reorder it such that `nums[0] < nums[1] > nums[2] < nums[3]....`

Example:

(1) Given `nums = [1, 5, 1, 1, 6, 4]` , one possible answer is `[1, 4, 1, 5, 1, 6]` .

(2) Given nums = `[1, 3, 2, 2, 3, 1]` , one possible answer is `[2, 3, 1, 3, 1, 2]` .

Note:

You may assume all input has valid answer.

Follow Up:

Can you do it in O(n) time and/or in-place with O(1) extra space?

Stack

- Stack

Stack

Basic Calculator

- Basic Calculator

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 + 1" = 2
2.  " 2-1 + 2 " = 3
3.  "(1+(4+5+2)-3)+(6+8)" = 23
4.  `
```

Note: Do not use the eval built-in library function.

Solution:

```
1.  public class Solution {
2.    public int calculate(String s) {
3.      char[] a = s.toCharArray();
4.
5.      Stack<Integer> stack = new Stack<Integer>();
6.      stack.push(1);
7.
```

```
 8.        int res = 0;
 9.        int sign = 1;
10.
11.        for (int i = 0; i < a.length; i++) {
12.          if (a[i] == ')') {
13.            stack.pop();
14.          } else if (a[i] == '+') {
15.            sign = 1;
16.          } else if (a[i] == '-') {
17.            sign = -1;
18.          } else if (a[i] == '(') {
19.            stack.push(stack.peek() * sign);
20.            sign = 1;
21.          } else if (Character.isDigit(a[i])) {
22.            int tmp = a[i] - '0';
23.
24.            while (i + 1 < s.length() && Character.isDigit(a[i + 1]))
25.              tmp = tmp * 10 + a[++i] - '0';
26.
27.            res += sign * stack.peek() * tmp;
28.          }
29.        }
30.
31.        return res;
32.      }
33. }
```

Evaluate Reverse Polish Notation

- Evaluate Reverse Polish Notation

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.

Some examples:

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

Solution:

```java
1.  public class Solution {
2.    public int evalRPN(String[] a) {
3.      Stack<Integer> stack = new Stack<Integer>();
4.
5.      for (int i = 0; i < a.length; i++) {
6.        switch (a[i]) {
7.          case "+":
8.            stack.push(stack.pop() + stack.pop());
9.            break;
10.
11.          case "-":
12.            stack.push(-stack.pop() + stack.pop());
13.            break;
14.
15.          case "*":
16.            stack.push(stack.pop() * stack.pop());
```

```
17.            break;
18.
19.        case "/":
20.            int n1 = stack.pop(), n2 = stack.pop();
21.            stack.push(n2 / n1);
22.            break;
23.
24.        default:
25.            stack.push(Integer.parseInt(a[i]));
26.      }
27.    }
28.
29.    return stack.pop();
30.  }
31. }
```

Flatten Nested List Iterator

- Flatten Nested List Iterator

Flatten Nested List Iterator

Given a nested list of integers, implement an iterator to flatten it.

Each element is either an integer, or a list — whose elements may also be integers or other lists.

Example 1:

Given the list `[[1,1],2,[1,1]]` ,

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: `[1,1,2,1,1]` .

Example 2:

Given the list `[1,[4,[6]]]` ,

By calling next repeatedly until hasNext returns false, the order of elements returned by next should be: `[1,4,6]` .

Implement Queue using Stacks

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.

Notes:

- You must use only standard operations of a stack — which means only `push to top`, `peek/pop from top`, `size`, and `is empty` operations are valid.
- Depending on your language, stack may not be supported natively. You may simulate a stack by using a list or deque (double-ended queue), as long as you use only standard operations of a stack.
- You may assume that all operations are valid (for example, no pop or peek operations will be called on an empty queue).

Solution:

```java
class MyQueue {
  Stack<Integer> s1 = new Stack<>();
  Stack<Integer> s2 = new Stack<>();

  // Push element x to the back of queue.
  public void push(int x) {
    s1.push(x);
  }

  // Removes the element from in front of queue.
  public void pop() {
    peek();
    s2.pop();
  }

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

    return s2.peek();
  }

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

Implement Stack using Queues

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.

Notes:

- You must use only standard operations of a queue — which means only push to back, peek/pop from front, size, and is empty operations are valid.
- Depending on your language, queue may not be supported natively. You may simulate a queue by using a list or deque (double-ended queue), as long as you use only standard operations of a queue.
- You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack).

Solution:

```
1.  class MyStack {
```

```
2.    Queue<Integer> queue = new LinkedList<Integer>();
3.
4.    // Push element x onto stack.
5.    public void push(int x) {
6.      queue.add(x);
7.      int k = queue.size();
8.      while (k-- > 1) {
9.        queue.add(queue.poll());
10.     }
11.   }
12.
13.   // Removes the element on top of the stack.
14.   public void pop() {
15.     queue.poll();
16.   }
17.
18.   // Get the top element.
19.   public int top() {
20.     return queue.peek();
21.   }
22.
23.   // Return whether the stack is empty.
24.   public boolean empty() {
25.     return queue.isEmpty();
26.   }
27. }
```

Largest Rectangle in Histogram

- Largest Rectangle in Histogram

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.
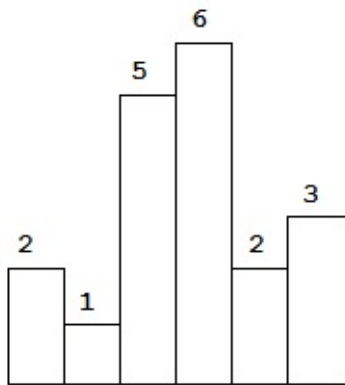
Above is a histogram where width of each bar is 1, given height = `[2,1,5,6,2,3]`.
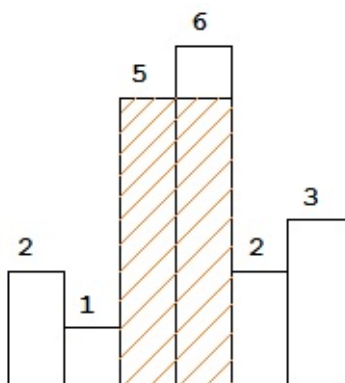
The largest rectangle is shown in the shaded area, which has area = `10` unit.

For example,

Given heights = `[2,1,5,6,2,3]` ,
return `10` .

Solution:

```java
public class Solution {
  public int largestRectangleArea(int[] h) {
    int n = h.length, i = 0, max = 0;

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

    while (i < n) {
      while (!s.isEmpty() && h[i] < h[s.peek()]) {
        max = Math.max(max, h[s.pop()] * (i - (s.isEmpty() ? 0 :
s.peek() + 1)));
      }
      s.push(i++);
    }

    while (!s.isEmpty()) {
      max = Math.max(max, h[s.pop()] * (n - (s.isEmpty() ? 0 :
s.peek() + 1)));
    }

    return max;
  }
}
```

Maximal Rectangle

- Maximal Rectangle

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.

Solution:

```
public class Solution {
  public int maximalRectangle(char[][] M) {
    if (M == null || M.length == 0 || M[0].length == 0) {
      return 0;
    }

    int max = 0;
    int m = M.length;
    int n = M[0].length;

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

    for (int i = 0; i < m; i++) {
      for (int j = 0; j < n; j++) {
        int v = getInt(M[i][j]);

        if (i == 0 || v == 0) {
          H[i][j] = v;
        } else {
          H[i][j] = v + H[i - 1][j];
        }
      }
    }

    int res = largestRectangleArea(H[i]);
```

```
25.
26.            max = Math.max(max, res);
27.        }
28.
29.        return max;
30.    }
31.
32.    public int largestRectangleArea(int[] A) {
33.        if (A == null) {
34.            return 0;
35.        }
36.
37.        Stack<Integer> stack = new Stack<Integer>();
38.
39.        int n = A.length, max = 0;
40.
41.        for (int i = 0; i < n; i++) {
42.            while (!stack.isEmpty() && A[i] <= A[stack.peek()]) {
43.                int index = stack.pop();
44.                int left = stack.isEmpty() ? 0 : stack.peek() + 1;
45.
46.                max = Math.max(max, A[index] * (i - left));
47.            }
48.
49.            stack.push(i);
50.        }
51.
52.        while (!stack.isEmpty()) {
53.            int index = stack.pop();
54.            int left = stack.isEmpty() ? 0 : stack.peek() + 1;
55.
56.            max = Math.max(max, A[index] * (n - left));
57.        }
58.
59.        return max;
60.    }
61.
62.    private int getInt(char ch) {
```

```
63.        return (int)(ch - '0');
64.    }
65. }
```

Min Stack

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.

Example:

```
1.  MinStack minStack = new MinStack();
2.  minStack.push(-2);
3.  minStack.push(0);
4.  minStack.push(-3);
5.  minStack.getMin();   --> Returns -3.
6.  minStack.pop();
7.  minStack.top();      --> Returns 0.
8.  minStack.getMin();   --> Returns -2.
```

Solution:

```
1.  class MinStack {
2.    Stack<Integer> s1 = new Stack<>();
3.    Stack<Integer[]> s2 = new Stack<>();
4.
5.    public void push(int x) {
6.      s1.push(x);
```

```
 7.        if (s2.isEmpty() || x < s2.peek()[0]) {
 8.          s2.push(new Integer[]{x, 1});
 9.        } else if (x == s2.peek()[0]) {
10.          s2.peek()[1]++;
11.        }
12.      }
13.
14.      public void pop() {
15.        if (s1.peek().intValue() == s2.peek()[0]) {
16.          s2.peek()[1]--;
17.          if (s2.peek()[1] == 0)
18.            s2.pop();
19.        }
20.        s1.pop();
21.      }
22.
23.      public int top() {
24.        return s1.peek();
25.      }
26.
27.      public int getMin() {
28.        return s2.peek()[0];
29.      }
30.    }
```

Remove Duplicate Letters

Remove Duplicate Letters

Given a string which contains only lowercase letters, remove duplicate letters so that every letter appear once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example:

Given `"bcabc"`
Return `"abc"`

Given `"cbacdcbc"`
Return `"acdb"`

Verify Preorder Serialization of a Binary Tree

Verify Preorder Serialization of a Binary Tree

One way to serialize a binary tree is to use preorder traversal. When we encounter a non-null node, we record the node's value. If it is a null node, we record using a sentinel value such as `#`.

```
1.      _9_
2.     /   \
3.    3     2
4.   / \   / \
5.  4   1 #   6
6. / \ / \   / \
7. # # # #   # #
```

For example, the above binary tree can be serialized to the string `"9,3,4,#,#,1,#,#,2,#,6,#,#"`, where `#` represents a null node.

Given a string of comma separated values, verify whether it is a correct preorder traversal serialization of a binary tree. Find an algorithm without reconstructing the tree.

Each comma separated value in the string must be either an integer or a character `'#'` representing `null` pointer.

You may assume that the input format is always valid, for example it could never contain two consecutive commas such as "1,,3".

Example 1:

`"9,3,4,#,#,1,#,#,2,#,6,#,#"`

Return `true`

Example 2:

`"1,#"`

Return `false`

Example 3:

`"9,#,#,1"`

Return `false`

String

- String

String

Add Binary

- Add Binary

Add Binary

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

For example,

a = `"11"`

b = `"1"`

Return `"100"` .

Solution:

```java
public class Solution {
    public String addBinary(String s1, String s2) {
        int i = s1.length() - 1, j = s2.length() - 1, c = 0;
        String s = "";

        while (i >= 0 || j >= 0 || c == 1) {
            int a = (i < 0) ? 0 : s1.charAt(i--) - '0';
            int b = (j < 0) ? 0 : s2.charAt(j--) - '0';

            s = (char)('0' + a ^ b ^ c) + s;
            c = (a + b + c) >> 1;
        }

        return s;
    }
}
```

Group Anagrams

* Group Anagrams

Group Anagrams

Given an array of strings, group anagrams together.

For example, given: `["eat", "tea", "tan", "ate", "nat", "bat"]` ,

Return:

```
1.  [
2.    ["ate", "eat","tea"],
3.    ["nat","tan"],
4.    ["bat"]
5.  ]
```

Note: All inputs will be in lower-case.

Solution:

```
1.  public class Solution {
2.    public List<String> anagrams(String[] strs) {
3.      List<String> res = new ArrayList<String>();
4.
5.      if (strs == null) {
6.        return res;
7.      }
8.
9.      Map<String, List<String>> map = new HashMap<String,
    List<String>>();
10.
11.      for (int i = 0; i < strs.length; i++) {
```

```
12.        String str = strs[i];
13.        String key = sort(str);
14.
15.        List<String> list = map.containsKey(key) ? map.get(key) : new
    ArrayList<String>();
16.        list.add(str);
17.
18.        map.put(key, list);
19.      }
20.
21.      for (Map.Entry<String, List<String>> entry : map.entrySet()) {
22.        String key = entry.getKey();
23.        List<String> list = entry.getValue();
24.
25.        if (list.size() > 1) {
26.          res.addAll(list);
27.        }
28.      }
29.
30.      return res;
31.    }
32.
33.    String sort(String str) {
34.      char[] chars = str.toCharArray();
35.      Arrays.sort(chars);
36.      return new String(chars);
37.    }
38.  }
```

Basic Calculator II

- Basic Calculator II

Basic Calculator II

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only non-negative integers, `+` , `-` , `*` , `/` operators and empty spaces . The integer division should truncate toward zero.

You may assume that the given expression is always valid.

Some examples:

```
1.  "3+2*2" = 7
2.  " 3/2 " = 1
3.  " 3+5 / 2 " = 5
```

Note: **Do not** use the `eval` built-in library function.

Solution:

```
1.  public class Solution {
2.    public int calculate(String s) {
3.      if (s == null || s.length() == 0)
4.        return 0;
5.
6.      s = "+" + s + "+";
```

```
  7.        char sign = '+';
  8.        char[] a = s.toCharArray();
  9.        int n = s.length(), res = 0, num = 0;
 10.        Stack<Integer> stack = new Stack<Integer>();
 11.
 12.        for (int i = 0; i < n; i++) {
 13.          if (Character.isDigit(a[i]))
 14.            num = num * 10 + a[i] - '0';
 15.
 16.          if (isOperator(a[i])) {
 17.            if (sign == '+') stack.push(num);
 18.            if (sign == '-') stack.push(-num);
 19.            if (sign == '*') stack.push(stack.pop() * num);
 20.            if (sign == '/') stack.push(stack.pop() / num);
 21.
 22.            num = 0;
 23.
 24.            sign = a[i];
 25.          }
 26.        }
 27.
 28.        for (int i : stack)
 29.          res += i;
 30.
 31.        return res;
 32.    }
 33.
 34.    boolean isOperator(char c) {
 35.      return c == '+' || c == '-' || c == '*' || c == '/';
 36.    }
 37.  }
```

Compare Version Numbers

Compare Version Numbers

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.
For instance, 2.5 is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.

Here is an example of version numbers ordering:

```
1. 0.1 < 1.1 < 1.2 < 13.37
```

Solution:

```
1. public class Solution {
2.   public int compareVersion(String version1, String version2) {
3.     char[] c1 = version1.toCharArray();
4.     char[] c2 = version2.toCharArray();
5.
6.     int i = 0, j = 0, v1 = 0, v2 = 0, n1 = c1.length, n2 = c2.length;
7.
8.     while (i < n1 && j < n2) {
```

```java
 9.         // get version number v1 and v2
10.         while (i < n1 && Character.isDigit(c1[i]))
11.           v1 = v1 * 10 + c1[i++] - '0';
12.
13.         while (j < n2 && Character.isDigit(c2[j]))
14.           v2 = v2 * 10 + c2[j++] - '0';
15.
16.         if (v1 > v2) return 1;
17.         if (v2 > v1) return -1;
18.
19.         // reset version numbers
20.         v1 = 0; v2 = 0;
21.
22.         // skip '.'
23.         i++; j++;
24.     }
25.
26.     while (i < n1 && Character.isDigit(c1[i]))
27.       v1 = v1 * 10 + c1[i++] - '0';
28.
29.     while (j < n2 && Character.isDigit(c2[j]))
30.       v2 = v2 * 10 + c2[j++] - '0';
31.
32.     if (v1 > v2) return 1;
33.     if (v1 < v2) return -1;
34.
35.     return 0;
36.   }
37. }
```

Count and Say

Count and Say

The count-and-say sequence is the sequence of integers beginning as follows:

```
1, 11, 21, 1211, 111221, ...
```

`1` is read off as `"one 1"` or `11` .

`11` is read off as `"two 1s"` or `21` .

`21` is read off as `"one 2, then one 1"` or `1211` .

Given an integer n, generate the nth sequence.

Note: The sequence of integers will be represented as a string.

Solution:

```
 1.  public class Solution {
 2.    public String countAndSay(int n) {
 3.      String t, s = "1";
 4.
 5.      while (n-- > 1) {
 6.        int c = 0;
 7.        t = "";
 8.
 9.        for (int i = 0; i <= s.length(); i++) {
10.          if (i == s.length() || (i > 0 && s.charAt(i) != s.charAt(i
     - 1))) {
11.            t += String.valueOf(c) + s.charAt(i - 1); c = 0; // say
```

```
12.            }
13.            c++; // count
14.        }
15.
16.        s = t;
17.    }
18.
19.    return s;
20.  }
21. }
```

Decode Ways

- Decode Ways

Decode Ways

A message containing letters from A-Z is being encoded to numbers using the following mapping:

```
1.  'A' -> 1
2.  'B' -> 2
3.  ...
4.  'Z' -> 26
```

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

For example,

Given encoded message `"12"`, it could be decoded as `"AB"` (1 2) or `"L"` (12).

The number of ways decoding `"12"` is 2.

Solution:

```
1.  public class Solution {
2.    public int numDecodings(String s) {
3.      if (s == null || s.length() == 0)
4.        return 0;
5.
6.      if (s.charAt(0) == '0')
7.        return 0;
8.
9.      int n = s.length();
```

```
10.
11.     int[] dp = new int[n + 1];
12.
13.     dp[0] = 1; dp[1] = 1;
14.
15.     for (int i = 2; i <= n; i++) {
16.       int count = 0;
17.
18.       if (s.charAt(i - 1) > '0')
19.         count = dp[i - 1];
20.
21.       if (s.charAt(i - 2) == '1' || (s.charAt(i - 2) == '2' &&
    s.charAt(i - 1) <= '6'))
22.           count += dp[i - 2];
23.
24.       dp[i] = count;
25.     }
26.
27.     return dp[n];
28.   }
29. }
```

Distinct Subsequences

- Distinct Subsequences

Distinct Subsequences

Given a string S and a string T, count the number of distinct subsequences of T in S.

A subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, `"ACE"` is a subsequence of `"ABCDE"` while `"AEC"` is not).

Here is an example:

S = `"rabbbit"` , T = `"rabbit"`

Return `3` .

Solution:

```java
public class Solution {
  public int numDistinct(String s, String t) {
    int n = s.length(), m = t.length();

    // dp(i, j) represents the count of S(1...i) and T(1...j)
    int[][] dp = new int[n + 1][m + 1];

    for (int i = 0; i <= n; i++)
      dp[i][0] = 1;

```

```
11.      for (int i = 1; i <= n; i++) {
12.        for (int j = 1; j <= m; j++) {
13.          if (s.charAt(i - 1) == t.charAt(j - 1)) {
14.            dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];
15.          } else {
16.            dp[i][j] = dp[i - 1][j];
17.          }
18.        }
19.      }
20.
21.      return dp[n][m];
22.    }
23. }
```

Edit Distance

- Edit Distance

Edit Distance

Given two words word1 and word2, find the minimum number of steps required to convert word1 to word2. (each operation is counted as 1 step.)

You have the following 3 operations permitted on a word:

1. Insert a character

2. Delete a character

3. Replace a character

Solution:

```java
public class Solution {
    public int minDistance(String word1, String word2) {
        int n1 = word1.length(), n2 = word2.length();

        int[][] dp = new int[n1 + 1][n2 + 1];

        for (int j = 1; j <= n2; j++) {
            dp[0][j] = j;
        }

        for (int i = 1; i <= n1; i++) {
            dp[i][0] = i;
        }

```

```
15.      for (int i = 1; i <= n1; i++) {
16.        for (int j = 1; j <= n2; j++) {
17.          int replace = dp[i - 1][j - 1] + ((word1.charAt(i - 1) ==
    word2.charAt(j - 1)) ? 0 : 1);
18.          int delete = dp[i - 1][j] + 1;
19.          int insert = dp[i][j - 1] + 1;
20.          dp[i][j] = Math.min(replace, Math.min(delete, insert));
21.        }
22.      }
23.
24.      return dp[n1][n2];
25.    }
26.  }
```

Encode and Decode Strings

- Encode and Decode Strings

Encode and Decode Strings

Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded back to the original list of strings.

Machine 1 (sender) has the function:

```
1.  string encode(vector<string> strs) {
2.    // ... your code
3.    return encoded_string;
4.  }
```

Machine 2 (receiver) has the function:

```
1.  vector<string> decode(string s) {
2.    //... your code
3.    return strs;
4.  }
```

So Machine 1 does:

```
1.  string encoded_string = encode(strs);
```

and Machine 2 does:

```
1.  vector<string> strs2 = decode(encoded_string);
```

`strs2` in Machine 2 should be the same as `strs` in Machine 1.

Implement the `encode` and `decode` methods.

Note:

- The string may contain any possible characters out of 256 valid ascii characters. Your algorithm should be generalized enough to work on any possible characters.
- Do not use class member/global/static variables to store states. Your encode and decode algorithms should be stateless.
- Do not rely on any library method such as eval or serialize methods. You should implement your own encode/decode algorithm.

Solution:

```java
1.  public class Codec {
2.
3.    // Encodes a list of strings to a single string.
4.    public String encode(List<String> strs) {
5.      StringBuilder sb = new StringBuilder();
6.      for(String s : strs) {
7.        sb.append(s.length()).append('/').append(s);
8.      }
9.      return sb.toString();
10.   }
11.
12.   // Decodes a single string to a list of strings.
13.   public List<String> decode(String s) {
14.     List<String> ret = new ArrayList<String>();
```

```
15.        int i = 0;
16.        while(i < s.length()) {
17.          int slash = s.indexOf('/', i);
18.          int size = Integer.valueOf(s.substring(i, slash));
19.          ret.add(s.substring(slash + 1, slash + size + 1));
20.          i = slash + size + 1;
21.        }
22.        return ret;
23.    }
24. }
25.
26. // Your Codec object will be instantiated and called as such:
27. // Codec codec = new Codec();
28. // codec.decode(codec.encode(strs));
```

Flip Game

-

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.

For example, given s = `"++++"` , after one move, it may become one of the following states:

```
1. [
2.    "--++",
3.    "+--+",
4.    "++--"
5. ]
```

If there is no valid move, return an empty list `[]` .

Solution:

```
1. public class Solution {
2.   public List<String> generatePossibleNextMoves(String s) {
3.     List<String> res = new ArrayList<>();
```

```
 4.
 5.      // sanity check
 6.      if (s == null || s.length() <= 1)
 7.        return res;
 8.
 9.      for (int i = 0; i < s.length() - 1; i++)
10.        if (s.startsWith("++", i))
11.          res.add(s.substring(0, i) + "--" + s.substring(i + 2));
12.
13.      return res;
14.    }
15.  }
```

Generate Parentheses

- Generate Parentheses

Generate Parentheses

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given n = 3, a solution set is:

```
1.  [
2.    "((()))",
3.    "(()())",
4.    "(())()",
5.    "()(())",
6.    "()()()"
7.  ]
```

Solution:

```
1.  public class Solution {
2.    public List<String> generateParenthesis(int n) {
3.      List<String> res = new ArrayList<String>();
4.      helper(n, 0, 0, "", res);
5.      return res;
6.    }
7.
8.    private void helper(int n, int left, int right, String sol,
    List<String> res) {
9.      if (right == n) {
10.       res.add(sol);
11.       return;
12.     }
```

```
13.
14.    if (left < n) {
15.      helper(n, left + 1, right, sol + "(", res);
16.    }
17.
18.    if (right < left) {
19.      helper(n, left, right + 1, sol + ")", res);
20.    }
21.  }
22. }
```

Group Shifted Strings

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:

```
1.  "abc" -> "bcd" -> ... -> "xyz"
```

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

For example, given: `["abc", "bcd", "acef", "xyz", "az", "ba", "a", "z"]` ,

A solution is:

```
1.  [
2.    ["abc","bcd","xyz"],
3.    ["az","ba"],
4.    ["acef"],
5.    ["a","z"]
6.  ]
```

Solution:

```
1.  public class Solution {
2.    public List<List<String>> groupStrings(String[] strings) {
3.      List<List<String>> res = new ArrayList<>();
```

```
4.        Map<String, List<String>> map = new HashMap<>();
5.
6.        for (int i = 0; i < strings.length; i++) {
7.          String s = strings[i];
8.          String key = hash(s);
9.
10.         List<String> list = map.containsKey(key) ? map.get(key) : new
    ArrayList<String>();
11.         list.add(s);
12.         map.put(key, list);
13.       }
14.
15.       for (List<String> list : map.values()) {
16.         Collections.sort(list);
17.         res.add(list);
18.       }
19.
20.       return res;
21.     }
22.
23.     String hash(String s) {
24.       String key = "";
25.
26.       for (int i = 1; i < s.length(); i++) {
27.         int diff = (int)(s.charAt(i) - s.charAt(i - 1));
28.         if (diff < 0) diff += 26;
29.         key += String.valueOf(diff);
30.       }
31.
32.       return key;
33.     }
34.  }
```

Integer to English Words

- Integer to English Words

Integer to English Words

Convert a non-negative integer to its english words representation. Given input is guaranteed to be less than 231 - 1.

For example,

```
1. 123 -> "One Hundred Twenty Three"
2. 12345 -> "Twelve Thousand Three Hundred Forty Five"
3. 1234567 -> "One Million Two Hundred Thirty Four Thousand Five
   Hundred Sixty Seven"
```

Hint:

- Did you see a pattern in dividing the number into chunk of words? For example, 123 and 123000.
- Group the number by thousands (3 digits). You can write a helper function that takes a number less than 1000 and convert just that chunk to words.
- There are many edge cases. What are some good test cases? Does your code work with input such as 0? Or 1000010? (middle chunk is zero and should not be printed out)

Solution:

```java
public class Solution {
  String[] bigs = "Hundred Thousand Million Billion".split(" ");
  String[] tens = "Twenty Thirty Forty Fifty Sixty Seventy Eighty Ninety".split(" ");
  String[] lows = "Zero One Two Three Four Five Six Seven Eight Nine Ten Eleven Twelve Thirteen Fourteen Fifteen Sixteen Seventeen Eighteen Nineteen".split(" ");

  public String numberToWords(int n) {
    if (n < 20)
      return lows[n];

    if (n < 100)
      return tens[n / 10 - 2] + helper(n % 10);

    if (n < 1000)
      return lows[n / 100] + " " + bigs[0] + helper(n % 100);

    int m = 1000;

    for (int i = 1; i < bigs.length; i++, m *= 1000)
      if (n / 1000 < m)
        return numberToWords(n / m) + " " + bigs[i] + helper(n % m);

    return numberToWords(n / m) + " " + bigs[bigs.length - 1] + helper(n % m);
  }

  public String helper(int n) {
    return n == 0 ? "" : " " + numberToWords(n);
  }
}
```

Integer to Roman

- Integer to Roman

Integer to Roman

Given an integer, convert it to a roman numeral.

Input is guaranteed to be within the range from 1 to 3999.

Solution:

```java
public class Solution {
  private char[][] romans = {
    {'I', 'V', 'X'},
    {'X', 'L', 'C'},
    {'C', 'D', 'M'},
    {'M', ' ', ' '}
  };

  public String intToRoman(int num) {
    StringBuilder sb = new StringBuilder();

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

    while (num > 0) {
      digits.add(num % 10);
      num /= 10;
    }

    int size = digits.size();

    for (int i = size - 1; i >= 0; i--) {
      String roman = getRoman(i, digits.get(i));
      sb.append(roman);
```

```
24.        }
25.
26.      return sb.toString();
27.    }
28.
29.    private String getRoman(int level, int digit) {
30.      char one  = romans[level][0];
31.      char five = romans[level][1];
32.      char ten  = romans[level][2];
33.
34.      StringBuilder sb = new StringBuilder();
35.
36.      if (digit == 9) {
37.        sb.append(one).append(ten);
38.      }
39.
40.      else if (digit >= 5) {
41.        sb.append(five);
42.        for (int i = 0; i < digit - 5; i++) {
43.          sb.append(one);
44.        }
45.      }
46.
47.      else if (digit == 4) {
48.        sb.append(one).append(five);
49.      }
50.
51.      else {
52.        for (int i = 0; i < digit; i++) {
53.          sb.append(one);
54.        }
55.      }
56.
57.      return sb.toString();
58.    }
59. }
```

Interleaving String

Interleaving String

Given s1, s2, s3, find whether s3 is formed by the interleaving of s1 and s2.

For example,

Given:

s1 = `"aabcc"` ,
s2 = `"dbbca"` ,

When s3 = `"aadbbcbcac"` , return `true` .

When s3 = `"aadbbbaccc"` , return `false` .

Solution:

```java
1.  public class Solution {
2.    public boolean isInterleave(String s1, String s2, String s3) {
3.      int n1 = s1.length(), n2 = s2.length(), n3 = s3.length();
4.
5.      if (n1 + n2 != n3)
6.        return false;
7.
8.      boolean[][] dp = new boolean[n1 + 1][n2 + 1];
9.
10.     dp[0][0] = true;
11.
12.     // first row
13.     for (int j = 1; j <= n2; j++) {
14.       dp[0][j] = (s2.charAt(j - 1) == s3.charAt(j - 1)) && dp[0][j
```

```
 - 1];
15.       }
16.
17.       // first col
18.       for (int i = 1; i <= n1; i++) {
19.           dp[i][0] = (s1.charAt(i - 1) == s3.charAt(i - 1)) && dp[i -
    1][0];
20.       }
21.
22.       // others
23.       for (int i = 1; i <= n1; i++) {
24.         for (int j = 1; j <= n2; j++) {
25.             dp[i][j] = ((s1.charAt(i - 1) == s3.charAt(i + j - 1)) &&
    dp[i - 1][j]) ||
26.                         ((s2.charAt(j - 1) == s3.charAt(i + j - 1)) &&
    dp[i][j - 1]);
27.         }
28.       }
29.
30.       return dp[n1][n2];
31.     }
32.  }
```

Length of Last Word

-

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.

Note: A word is defined as a character sequence consists of non-space characters only.

For example,

Given s = `"Hello World"`,

return `5`.

Solution:

```java
public class Solution {
  public int lengthOfLastWord(String s) {
    if (s == null)
      return 0;

    int i = s.length() - 1;

    while (i >= 0 && s.charAt(i) == ' ') { i--; }

    int j = i;

    while (i >= 0 && s.charAt(i) != ' ') { i--; }
```

```
13.
14.      return j - i;
15.    }
16.  }
```

Letter Combinations of a Phone Number

- Letter Combinations of a Phone Number

Letter Combinations of a Phone Number

Given a digit string, return all possible letter combinations that the number could represent.

A mapping of digit to letters (just like on the telephone buttons) is given below.


```
1. Input:Digit string "23"
2. Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
```

Note:

Although the above answer is in lexicographical order, your answer could be in any order you want.

Solution:

```java
1. public class Solution {
2.   public List<String> letterCombinations(String digits) {
3.     String[] keyboard = {" ", "", "abc", "def", "ghi", "jkl",
   "mno", "pqrs", "tuv", "wxyz"};
4.
```

```
 5.        List<String> res = new ArrayList<String>();
 6.
 7.        helper(digits, keyboard, 0, "", res);
 8.
 9.        return res;
10.    }
11.
12.    private void helper(String digits, String[] keyboard, int index,
    String sol, List<String> res) {
13.        if (sol.length() == digits.length()) {
14.            res.add(sol);
15.            return;
16.        }
17.
18.        int idx = (int)(digits.charAt(index) - '0');
19.        String key = keyboard[idx];
20.
21.        for (int i = 0; i < key.length(); i++) {
22.          helper(digits, keyboard, index + 1, sol + key.charAt(i),
    res);
23.        }
24.    }
25. }
```

Longest Common Prefix

- Longest Common Prefix

Longest Common Prefix

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

Solution:

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

    for (int i = 0; i < strs[0].length(); i++) {
      for (int j = 1; j < strs.length; j++) {
        if (i >= strs[j].length() || strs[j].charAt(i) !=
    strs[0].charAt(i)) {
          return strs[0].substring(0, i);
        }
      }
    }

    return strs[0];
  }
}
```

Longest Palindromic Substring

- Longest Palindromic Substring

Longest Palindromic Substring

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

Solution:

```java
public class Solution {
  public String longestPalindrome(String s) {
    int n = s.length();

    String res = null;

    boolean[][] dp = new boolean[n][n];

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

        if (dp[i][j] && (res == null || j - i + 1 > res.length()))
{
          res = s.substring(i, j + 1);
        }
      }
    }

    return res;
  }
```

```
21.  }
```

Longest Substring with At Most K Distinct Characters

-

Longest Substring with At Most K Distinct Characters

Given a string, find the length of the longest substring T that contains at most k distinct characters.

For example, Given s = `"eceba"` and k = 2,

T is `"ece"` which its length is 3.

Longest Valid Parentheses

- Longest Valid Parentheses

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`.

Solution:

```
 1.  public class Solution {
 2.    public int longestValidParentheses(String s) {
 3.        int i = -1, j = 0, max = 0;
 4.
 5.        Stack<Integer> stack = new Stack<>();
 6.
 7.        while (j < s.length()) {
 8.        char c = s.charAt(j);
 9.
10.        if (c == '(') {
11.          stack.push(j);
12.        } else {
13.          if (stack.isEmpty()) {
14.            i = j;
15.          } else {
```

```
16.             stack.pop();
17.             max = Math.max(max, j - (stack.isEmpty() ? i :
    stack.peek()));
18.           }
19.         }
20.
21.       j++;
22.     }
23.
24.     return max;
25.   }
26. }
```

Multiply Strings

- Multiply Strings

Multiply Strings

Given two numbers represented as strings, return multiplication of the numbers as a string.

Note:

- The numbers can be arbitrarily large and are non-negative.
- Converting the input string to integer is **NOT** allowed.
- You should **NOT** use internal library such as **BigInteger**.

Solution:

```java
public class Solution {
  public String multiply(String num1, String num2) {
    if (num1 == null || num2 == null || num1.length() == 0 ||
num2.length() == 0) {
      return "";
    }

    char[] chs1 = num1.toCharArray();
    char[] chs2 = num2.toCharArray();

    reverse(chs1, 0, chs1.length - 1);
    reverse(chs2, 0, chs2.length - 1);

    int n = chs1.length;
```

```
14.      int m = chs2.length;
15.
16.      // total length won't be longer than m + n
17.      int[] res = new int[m + n];
18.
19.      for (int i = 0; i < n; i++) {
20.        int val = 0;
21.        int d1 = (int)(chs1[i] - '0');
22.
23.        for (int j = 0; j < m; j++) {
24.          int d2 = (int)(chs2[j] - '0');
25.
26.          val = d1 * d2 + res[i + j] + val;
27.          res[i + j] = val % 10;
28.          val /= 10;
29.        }
30.
31.        if (val > 0) {
32.          res[i + m] = val;
33.        }
34.      }
35.
36.      StringBuilder sb = new StringBuilder();
37.      boolean valid = false;
38.
39.      for (int i = res.length - 1; i >= 0; i--) {
40.        // ignore leading zeros
41.        if (!valid && res[i] == 0) {
42.          continue;
43.        }
44.
45.        valid = true;
46.        sb.append(res[i]);
47.      }
48.
49.      if (sb.length() == 0) {
50.          return "0";
51.      }
```

```
52.
53.      return sb.toString();
54.    }
55.
56.    void reverse(char[] chs, int lo, int hi) {
57.      while (lo < hi) {
58.        swap(chs, lo++, hi--);
59.      }
60.    }
61.
62.    void swap(char[] chs, int i, int j) {
63.      char ch = chs[i];
64.      chs[i] = chs[j];
65.      chs[j] = ch;
66.    }
67. }
```

One Edit Distance

- One Edit Distance

One Edit Distance

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

Solution:

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

    if (s.length() > t.length())
      return isOneEditDistance(t, s);

    int i = 0;

    while (i < s.length() && i < t.length()) {
      if (s.charAt(i) != t.charAt(i)) {
        return s.substring(i + 1).equals(t.substring(i + 1)) ||
               s.substring(i).equals(t.substring(i + 1));
      }

      i++;
    }

    return t.length() - i == 1;
  }
}
```

Read N Characters Given Read4 II - Call multiple times

Read N Characters Given Read4 II - Call multiple times

The API: `int read4(char *buf)` reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

By using the `read4` API, implement the function `int read(char *buf, int n)` that reads n characters from the file.

Note:

The read function may be called multiple times.

Solution:

```
1.  /* The read4 API is defined in the parent class Reader4.
2.      int read4(char[] buf); */
3.
4.  public class Solution extends Reader4 {
5.    // a pointer in the buffer
6.    int ptr = 0;
7.    // how many left in the buffer after last call
```

```
 8.    int left = 0;
 9.    // as read() can be called multiple times
10.    // we should only allocate the buffer once
11.    char[] buffer = new char[4];
12.
13.    /**
14.     * @param buf Destination buffer
15.     * @param n   Maximum number of characters to read
16.     * @return    The number of characters read
17.     */
18.    public int read(char[] buf, int n) {
19.      // end of file flag
20.      boolean eof = false;
21.      // total bytes have been read this time
22.      int total = 0;
23.
24.      while (!eof && total < n) {
25.        // if we still have some leftovers, use them
26.        // otherwise we read another 4 chars
27.        int size  = (left > 0) ? left : read4(buffer);
28.
29.        // check if it's the end of the file
30.        eof = (left == 0 && size < 4);
31.
32.        // get the actual count we are going to read
33.        int count = Math.min(size, n - total);
34.
35.        // update the count of leftovers
36.        left = size - count;
37.
38.        // copy
39.        for (int i = 0; i < count; i++)
40.          buf[total++] = buffer[ptr + i];
41.
42.        // update the pointer
43.        ptr = (ptr + count) % 4;
44.      }
45.
```

```
46.        return total;
47.    }
48. }
```

Read N Characters Given Read4

Read N Characters Given Read4

The API: `int read4(char *buf)` reads 4 characters at a time from a file.

The return value is the actual number of characters read. For example, it returns 3 if there is only 3 characters left in the file.

By using the `read4` API, implement the function `int read(char *buf, int n)` that reads n characters from the file.

Note:

The `read` function will only be called once for each test case.

Solution:

```
1.  /* The read4 API is defined in the parent class Reader4.
2.       int read4(char[] buf); */
3.
4.  public class Solution extends Reader4 {
5.    /**
6.     * @param buf Destination buffer
7.     * @param n   Maximum number of characters to read
8.     * @return    The number of characters read
9.     */
10.   public int read(char[] buf, int n) {
```

```
11.      boolean eof = false;      // end of file flag
12.      int total = 0;            // total bytes have read
13.      char[] tmp = new char[4]; // temp buffer
14.
15.      while (!eof && total < n) {
16.        int count = read4(tmp);
17.
18.        // check if it's the end of the file
19.        eof = count < 4;
20.
21.        // get the actual count
22.        count = Math.min(count, n - total);
23.
24.        // copy from temp buffer to buf
25.        for (int i = 0; i < count; i++)
26.          buf[total++] = tmp[i];
27.      }
28.
29.      return total;
30.    }
31. }
```

Regular Expression Matching

Regular Expression Matching

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

```
1.  '.' Matches any single character.
2.  '*' Matches zero or more of the preceding element.
3.
4.  The matching should cover the entire input string (not partial).
5.
6.  The function prototype should be:
7.  bool isMatch(const char *s, const char *p)
8.
9.  Some examples:
10. isMatch("aa","a") → false
11. isMatch("aa","aa") → true
12. isMatch("aaa","aa") → false
13. isMatch("aa", "a*") → true
14. isMatch("aa", ".*") → true
15. isMatch("ab", ".*") → true
16. isMatch("aab", "c*a*b") → true
```

Solution:

```
1.  public class Solution {
2.    public boolean isMatch(String s, String p) {
3.      int n = s.length(), m = p.length();
4.
5.      boolean[][] dp = new boolean[n + 1][m + 1];
6.
7.      // initialize
```

```
8.        dp[0][0] = true;
9.
10.       for (int j = 1; j <= m; j++)
11.         dp[0][j] = j > 1 && p.charAt(j - 1) == '*' && dp[0][j - 2];
12.
13.       for (int i = 1; i <= n; i++) {
14.         for (int j = 1; j <= m; j++) {
15.           if (p.charAt(j - 1) != '*')
16.             dp[i][j] = dp[i - 1][j - 1] && isMatch(s.charAt(i - 1),
    p.charAt(j - 1));
17.           else
18.             dp[i][j] = dp[i][j - 2] || dp[i - 1][j] &&
    isMatch(s.charAt(i - 1), p.charAt(j - 2));
19.         }
20.       }
21.
22.       return dp[n][m];
23.     }
24.
25.     boolean isMatch(char a, char b) {
26.       return a == b || b == '.';
27.     }
28. }
```

Restore IP Addresses

- Restore IP Addresses

Restore IP Addresses

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:

Given `"25525511135"`,

return `["255.255.11.135", "255.255.111.35"]`. (Order does not matter)

Solution:

```
1.  public class Solution {
2.    public List<String> restoreIpAddresses(String s) {
3.      List<String> res = new ArrayList<String>();
4.      helper(s, 0, 0, "", res);
5.      return res;
6.    }
7.
8.    void helper(String s, int index, int segment, String sol,
      List<String> res) {
9.      if (segment == 4 && index == s.length()) {
10.       res.add(sol);
11.       return;
12.     }
13.
14.     // too many characters left
15.     if (s.length() > index + (4 - segment) * 3) {
```

```
16.          return;
17.        }
18.
19.        // not enough characters left
20.        if (s.length() < index + (4 - segment)) {
21.          return;
22.        }
23.
24.        for (int i = index; i < index + 3 && i < s.length(); i++) {
25.          String str = s.substring(index, i + 1);
26.          int num = Integer.parseInt(str);
27.
28.          if ((s.charAt(index) == '0' && i > index) || (num > 255)) {
29.            return;
30.          }
31.
32.          helper(s, i + 1, segment + 1, sol + str + (segment == 3 ? ""
    : "."), res);
33.        }
34.      }
35.  }
```

Reverse Words in a String II

- Reverse Words in a String II

Reverse Words in a String II

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

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

For example,

Given s = `"the sky is blue"` ,

return `"blue is sky the"` .

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

Related problem: Rotate Array

Solution:

```
1.  public class Solution {
2.    public void reverseWords(char[] chars) {
3.      int n = chars.length;
4.
5.      // step 1. reverse the whole string
6.      reverse(chars, 0, n - 1);
7.
```

```
8.        // step 2. reverse each word
9.        int i = 0, j = 0;
10.
11.       while (i < n) {
12.          // trim the space in front
13.          while (i < n && chars[i] == ' ') { i++; }
14.          j = i;
15.
16.          // find the border
17.          while (j < n && chars[j] != ' ') { j++; }
18.
19.          // reverse from i to j - 1
20.          reverse(chars, i, j - 1);
21.
22.          // continue
23.          i = j;
24.       }
25.    }
26.
27.    private void reverse(char[] chars, int start, int end) {
28.       while (start < end) {
29.          char tmp = chars[start];
30.          chars[start] = chars[end];
31.          chars[end] = tmp;
32.
33.          start++;
34.          end--;
35.       }
36.    }
37. }
```

Reverse Words in a String

- Reverse Words in a String

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"` .

Solution:

```
public class Solution {
  public String reverseWords(String s) {
    if (s == null) return null;

    char[] a = s.toCharArray();
    int n = a.length;

    // step 1. reverse the whole string
    reverse(a, 0, n - 1);

    // step 2. reverse each word
    reverseWords(a, n);

    // step 3. clean multiple spaces
    return cleanSpaces(a, n);
  }

  void reverseWords(char[] a, int n) {
    int i = 0, j = 0;
```

```
20.
21.     while (i < n) {
22.       while (i < j || i < n && a[i] == ' ') i++; // skip spaces
23.       while (j < i || j < n && a[j] != ' ') j++; // skip non spaces
24.       reverse(a, i, j - 1);                       // reverse the
    word
25.     }
26.   }
27.
28.   // trim leading, trailing and multiple spaces
29.   String cleanSpaces(char[] a, int n) {
30.     int i = 0, j = 0;
31.
32.     while (j < n) {
33.       while (j < n && a[j] == ' ') j++;           // skip spaces
34.       while (j < n && a[j] != ' ') a[i++] = a[j++]; // keep non
    spaces
35.       while (j < n && a[j] == ' ') j++;           // skip spaces
36.       if (j < n) a[i++] = ' ';                    // keep only
    one space
37.     }
38.
39.     return new String(a).substring(0, i);
40.   }
41.
42.   // reverse a[] from a[i] to a[j]
43.   private void reverse(char[] a, int i, int j) {
44.     while (i < j) {
45.       char t = a[i];
46.       a[i++] = a[j];
47.       a[j--] = t;
48.     }
49.   }
50. }
```

Roman to Integer

- Roman to Integer

Roman to Integer

Given a roman numeral, convert it to an integer.

Input is guaranteed to be within the range from 1 to 3999.

Solution:

```java
public class Solution {
  public int romanToInt(String s) {
    // Roman to Int map
    Map<Character, Integer> map = new HashMap<>();
    map.put('I', 1);
    map.put('V', 5);
    map.put('X', 10);
    map.put('L', 50);
    map.put('C', 100);
    map.put('D', 500);
    map.put('M', 1000);

    int n = s.length();
    int i = n - 1;
    int sum = map.get(s.charAt(i--));

    while (i >= 0) {
      int curr = map.get(s.charAt(i));
      int next = map.get(s.charAt(i + 1));

      if (curr >= next) {
        sum += curr;
      } else {
```

```
24.            sum -= curr;
25.        }
26.
27.        i--;
28.    }
29.
30.    return sum;
31.    }
32. }
```

Scramble String

-

Scramble String

Given a string s1, we may represent it as a binary
tree by partitioning it to two non-empty substrings
recursively.

Below is one possible representation of s1 =
`"great"` :

```
1.      great
2.     /    \
3.   gr    eat
4.  / \    / \
5. g  r   e  at
6.           / \
7.          a  t
```

To scramble the string, we may choose any non-leaf
node and swap its two children.

For example, if we choose the node `"gr"` and swap
its two children, it produces a scrambled string
`"rgeat"` .

```
1.      rgeat
2.     /    \
3.   rg    eat
4.  / \    / \
5. r  g   e  at
6.           / \
```

```
7.          a   t
```

We say that `"rgeat"` is a scrambled string of `"great"` .

Similarly, if we continue to swap the children of nodes `"eat"` and `"at"` , it produces a scrambled string `"rgtae"` .

```
1.       rgtae
2.      /    \
3.    rg    tae
4.   / \    / \
5.  r   g  ta  e
6.          / \
7.         t   a
```

We say that `"rgtae"` is a scrambled string of `"great"` .

Given two strings s1 and s2 of the same length, determine if s2 is a scrambled string of s1.

Solution:

```java
1.  public class Solution {
2.    public boolean isScramble(String s1, String s2) {
3.      if (!isAnagram(s1, s2))
4.        return false;
5.
6.      if (s1.equals(s2))
7.        return true;
8.
9.      int n = s1.length();
10.
11.     for (int i = 1; i < n; i++) {
```

```
12.        String s11 = s1.substring(0, i);
13.        String s12 = s1.substring(i, n);
14.
15.        String s21 = s2.substring(0, i);
16.        String s22 = s2.substring(i, n);
17.
18.        if (isScramble(s11, s21) && isScramble(s12, s22))
19.          return true;
20.
21.        s21 = s2.substring(0, n - i);
22.        s22 = s2.substring(n - i, n);
23.
24.        if (isScramble(s11, s22) && isScramble(s12, s21))
25.          return true;
26.      }
27.
28.      return false;
29.    }
30.
31.    boolean isAnagram(String s1, String s2) {
32.      if (s1 == null || s2 == null || s1.length() != s2.length())
33.        return false;
34.
35.      int[] arr = new int[26];
36.
37.      for (int i = 0; i < s1.length(); i++) {
38.        arr[s1.charAt(i) - 'a']++;
39.        arr[s2.charAt(i) - 'a']--;
40.      }
41.
42.      for (int i = 0; i < 26; i++)
43.        if (arr[i] != 0) return false;
44.
45.      return true;
46.    }
47. }
```

Shortest Palindrome

- Shortest Palindrome

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"`.

Solution:

```java
public class Solution {
  public String shortestPalindrome(String s) {
    String r = new StringBuilder(s).reverse().toString();
    int[] lps = getLPS(s + '|' + r);
    return r.substring(0, r.length() - lps[lps.length - 1]) + s;
  }

  // KMP get longest prefix and suffix count
  int[] getLPS(String s) {
    int[] lps = new int[s.length()];
    int i = 1, len = 0;

    while (i < s.length()) {
      if (s.charAt(i) == s.charAt(len)) lps[i++] = ++len;
      else if (len == 0)                lps[i++] = 0;
      else                              len = lps[len - 1];
```

```
17.        }
18.
19.     return lps;
20.    }
21. }
```

Simplify Path

- Simplify Path

Simplify Path

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

For example,

path = `"/home/"``, => `"/home"``

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

Corner Cases:

- Did you consider the case where path = `"/../"` ?
  In this case, you should return `"/"` .
- Another corner case is the path might contain multiple slashes `'/'` together, such as `"/home//foo/"` .
  In this case, you should ignore redundant slashes and return `"/home/foo"` .

Solution:

```
1.  public class Solution {
2.    public String simplifyPath(String path) {
3.      Set<String> set = new HashSet<>(Arrays.asList("", ".", ".."));
4.      Deque<String> deque = new ArrayDeque<>();
5.
6.      for (String token : path.split("/")) {
7.        if (token.equals("..") && !deque.isEmpty())
```

```
8.          deque.pollLast();
9.
10.       if (set.contains(token))
11.         continue;
12.
13.       deque.addLast(token);
14.     }
15.
16.     StringBuilder sb = new StringBuilder();
17.
18.     while (!deque.isEmpty()) {
19.       sb.append("/" + deque.pollFirst());
20.     }
21.
22.     return sb.length() == 0 ? "/" : sb.toString();
23.   }
24. }
```

String to Integer

- String to Integer

String to Integer

Implement atoi to convert a string to an integer.

Hint:

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

Notes:

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

Solution:

```java
public class Solution {
  public int myAtoi(String str) {
    if (str == null) {
      return 0;
    }

    // trim the leading and trailing spaces
    str = str.trim();

    if (str.length() == 0) {
      return 0;
```

```
12.        }
13.
14.        char[] chars = str.toCharArray();
15.
16.        boolean positive = true;
17.
18.        int i = 0;
19.
20.        if (chars[0] == '-') {
21.          positive = false;
22.          i = 1;
23.        }
24.
25.        if (chars[0] == '+') {
26.          i = 1;
27.        }
28.
29.        int num = 0;
30.
31.        while (i < chars.length) {
32.          char ch = chars[i++];
33.
34.          if (ch < '0' || ch > '9') {
35.            return num;
36.          }
37.
38.          int d = (int)(ch - '0');
39.
40.          if (positive) {
41.            if (num > (Integer.MAX_VALUE - d) / 10) {
42.              return Integer.MAX_VALUE;
43.            }
44.            num = num * 10 + d;
45.          } else {
46.            if (num < (Integer.MIN_VALUE + d) / 10) {
47.              return Integer.MIN_VALUE;
48.            }
49.            num = num * 10 - d;
```

```
50.          }
51.       }
52.
53.      return num;
54.    }
55.  }
```

Text Justification

Text Justification

Given an array of words and a length L, format the text such that each line has exactly L characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly L characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line do not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left justified and no extra space is inserted between words.

For example,

words: `["This", "is", "an", "example", "of", "text", "justification."]`

L: `16` .

Return the formatted lines as:

```
1.  [
2.    "This    is    an",
3.    "example  of text",
4.    "justification.  "
5.  ]
```

Note: Each word is guaranteed not to exceed L in length.

Solution:

```java
1.  public class Solution {
2.    public List<String> fullJustify(String[] words, int maxWidth) {
3.      List<String> res = new ArrayList<>();
4.
5.      int n = words.length, i = 0;
6.
7.      while (i < n) {
8.        int j = i, len = -1;
9.
10.       while (j < n && len + words[j].length() + 1 <= maxWidth) {
11.         len += words[j].length() + 1;
12.         j++;
13.       }
14.
15.       int space = 1, extra = 0;
16.
17.       if (j != i + 1 && j != n) {
18.         space = (maxWidth - len) / (j - i - 1) + 1;
19.         extra = (maxWidth - len) % (j - i - 1);
20.       }
21.
22.       StringBuilder sb = new StringBuilder();
23.
24.       sb.append(words[i]);
```

```
25.
26.        for (int k = i + 1; k < j; k++) {
27.          for (int s = space; s > 0; s--) {
28.            sb.append(' ');
29.          }
30.
31.          if (extra-- > 0) {
32.            sb.append(' ');
33.          }
34.
35.          sb.append(words[k]);
36.        }
37.
38.        int left = maxWidth - sb.length();
39.
40.        while (left-- > 0) {
41.          sb.append(' ');
42.        }
43.
44.        res.add(sb.toString());
45.
46.        i = j;
47.      }
48.
49.      return res;
50.    }
51.  }
```

Valid Number

- Valid Number

Valid Number

Validate if a given string is numeric.

Some examples:

`"0"` => `true`

`" 0.1 "` => `true`

`"abc"` => `false`

`"1 a"` => `false`

`"2e10"` => `true`

Note:

It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one.

Solution:

```
1.  public class Solution {
2.    public boolean isNumber(String s) {
3.      if (s == null) return false;
4.
5.      s = s.trim();
6.      int n = s.length();
7.
8.      if (n == 0) return false;
```

```
 9.
10.     // Define flags
11.     int signCount = 0;
12.
13.     boolean hasE = false;
14.     boolean hasNum = false;
15.     boolean hasPoint = false;
16.
17.     // Go through the characters
18.     for (int i = 0; i < n; i++) {
19.       char c = s.charAt(i);
20.
21.       // invalid character
22.       if (!isValid(c)) return false;
23.
24.       // digit
25.       if (c >= '0' && c <= '9') hasNum = true;
26.
27.       // e or E
28.       if (c == 'e' || c == 'E') {
29.         // e 之前一定要有数字
30.         if (hasE || !hasNum) return false;
31.         // e 不能作为最后一个
32.         if (i == n - 1) return false;
33.
34.         hasE = true;
35.       }
36.
37.       // decimal place
38.       if (c == '.') {
39.         // 小数点不能出现在 e 之后
40.         if (hasPoint || hasE) return false;
41.         // 小数点如果在最后一位出现，那么前面必须有要数字
42.         if (i == n - 1 && !hasNum) return false;
43.
44.         hasPoint = true;
45.       }
46.
```

```
47.        // signs
48.        if (c == '+' || c == '-') {
49.            // 不允许超过两个符号出现
50.            if (signCount == 2) return false;
51.            // 不允许符号在最后出现
52.            if (i == n - 1) return false;
53.            // 符号出现在中间的前提是前面有 e
54.            if (i > 0 && !hasE) return false;
55.
56.            signCount++;
57.        }
58.    }
59.
60.    return true;
61.    }
62.
63.    boolean isValid(char c) {
64.        return c == '.' || c == '+' || c == '-' || c == 'e' || c == 'E'
    || c >= '0' && c <= '9';
65.    }
66. }
```

Valid Parentheses

- Valid Parentheses

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.

Solution:

```
public class Solution {
  public boolean isValid(String s) {
    char[] p = "(){}[]".toCharArray();
    Map<Character, Character> map = new HashMap<>();

    for (int i = 0; i < p.length - 1; i += 2)
      map.put(p[i + 1], p[i]);

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

    for (char c : s.toCharArray()) {
      if (c == '(' || c == '{' || c == '[')
        stack.push(c);
      else if (stack.isEmpty() || stack.peek() != map.get(c))
        return false;
      else
        stack.pop();
    }

```

```
20.      return stack.isEmpty();
21.   }
22. }
```

Wildcard Matching

- Wildcard Matching

Wildcard Matching

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

```
1.  '?' Matches any single character.
2.  '*' Matches any sequence of characters (including the empty
    sequence).
3.
4.  The matching should cover the entire input string (not partial).
5.
6.  The function prototype should be:
7.  bool isMatch(const char *s, const char *p)
8.
9.  Some examples:
10. isMatch("aa","a") → false
11. isMatch("aa","aa") → true
12. isMatch("aaa","aa") → false
13. isMatch("aa", "*") → true
14. isMatch("aa", "a*") → true
15. isMatch("ab", "?*") → true
16. isMatch("aab", "c*a*b") → false
```

Solution:

```
1.  public class Solution {
2.    public boolean isMatch(String s, String p) {
3.      int n = s.length(), m = p.length();
4.
5.      boolean[][] dp = new boolean[n + 1][m + 1];
6.
```

```
 7.        // initialize
 8.        dp[0][0] = true;
 9.
10.        for (int j = 1; j <= m; j++) {
11.          dp[0][j] = dp[0][j - 1] && p.charAt(j - 1) == '*';
12.        }
13.
14.        for (int i = 1; i <= n; i++) {
15.          for (int j = 1; j <= m; j++) {
16.            if (p.charAt(j - 1) != '*') {
17.              dp[i][j] = dp[i - 1][j - 1] && (s.charAt(i - 1) ==
    p.charAt(j - 1) || p.charAt(j - 1) == '?');
18.            } else {
19.              dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
20.            }
21.          }
22.        }
23.
24.        return dp[n][m];
25.    }
26. }
```

Word Ladder II

- Word Ladder II

Word Ladder II

Given two words (beginWord and endWord), and a dictionary's word list, find all shortest transformation sequence(s) from beginWord to endWord, such that:

1. Only one letter can be changed at a time
2. Each intermediate word must exist in the word list

For example,

Given:

beginWord = `"hit"`

endWord = `"cog"`

wordList = `["hot","dot","dog","lot","log"]`

Return

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

Note:

- All words have the same length.
- All words contain only lowercase alphabetic characters.

Solution:

```java
public class Solution {
  public List<List<String>> findLadders(String start, String end,
Set<String> dict) {
    // hash set for both ends
    Set<String> set1 = new HashSet<String>();
    Set<String> set2 = new HashSet<String>();

    // initial words in both ends
    set1.add(start);
    set2.add(end);

    // we use a map to help construct the final result
    Map<String, List<String>> map = new HashMap<String,
List<String>>();

    // build the map
    helper(dict, set1, set2, map, false);

    List<List<String>> res = new ArrayList<List<String>>();
    List<String> sol = new ArrayList<String>(Arrays.asList(start));

    // recursively build the final result
    generateList(start, end, map, sol, res);

    return res;
  }

  boolean helper(Set<String> dict, Set<String> set1, Set<String>
set2, Map<String, List<String>> map, boolean flip) {
    if (set1.isEmpty()) {
      return false;
```

```
29.        }
30.
31.        if (set1.size() > set2.size()) {
32.          return helper(dict, set2, set1, map, !flip);
33.        }
34.
35.        // remove words on current both ends from the dict
36.        dict.removeAll(set1);
37.        dict.removeAll(set2);
38.
39.        // as we only need the shortest paths
40.        // we use a boolean value help early termination
41.        boolean done = false;
42.
43.        // set for the next level
44.        Set<String> set = new HashSet<String>();
45.
46.        // for each string in end 1
47.        for (String str : set1) {
48.          for (int i = 0; i < str.length(); i++) {
49.            char[] chars = str.toCharArray();
50.
51.            // change one character for every position
52.            for (char ch = 'a'; ch <= 'z'; ch++) {
53.              chars[i] = ch;
54.
55.              String word = new String(chars);
56.
57.              // make sure we construct the tree in the correct
   direction
58.              String key = flip ? word : str;
59.              String val = flip ? str : word;
60.
61.              List<String> list = map.containsKey(key) ? map.get(key) :
   new ArrayList<String>();
62.
63.              if (set2.contains(word)) {
64.                done = true;
```

```
65.
66.                list.add(val);
67.                map.put(key, list);
68.              }
69.
70.            if (dict.contains(word)) {
71.                set.add(word);
72.
73.                list.add(val);
74.                map.put(key, list);
75.              }
76.            }
77.          }
78.        }
79.
80.        // early terminate if done is true
81.        return done || helper(dict, set2, set, map, !flip);
82.      }
83.
84.      void generateList(String start, String end, Map<String,
      List<String>> map, List<String> sol, List<List<String>> res) {
85.        if (start.equals(end)) {
86.          res.add(new ArrayList<String>(sol));
87.          return;
88.        }
89.
90.        // need this check in case the diff between start and end
      happens to be one
91.        // e.g "a", "c", {"a", "b", "c"}
92.        if (!map.containsKey(start)) {
93.          return;
94.        }
95.
96.        for (String word : map.get(start)) {
97.          sol.add(word);
98.          generateList(word, end, map, sol, res);
99.          sol.remove(sol.size() - 1);
100.        }
```

```
101.     }
102. }
```

ZigZag Conversion

- ZigZag Conversion

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)

```
1.  P   A   H   N
2.  A P L S I I G
3.  Y   I   R
```

And then read line by line: `"PAHNAPLSIIGYIR"`

Write the code that will take a string and make this conversion given a number of rows:

```
1.  string convert(string text, int nRows);
```

`convert("PAYPALISHIRING", 3)` should return `"PAHNAPLSIIGYIR"` .

Solution:

```
1.  public class Solution {
2.    public String convert(String s, int nRows) {
3.      if (nRows == 1)
4.        return s;
5.
6.      List<StringBuilder> list = new ArrayList<>();
7.
8.      for (int i = 0; i < nRows; i++) {
```

```
 9.         list.add(new StringBuilder());
10.      }
11.
12.      int row = 0, dir = 0;
13.      for (int i = 0; i < s.length(); i++) {
14.        list.get(row).append(s.charAt(i));
15.
16.        if (row == 0) dir = 1;
17.        if (row == nRows - 1) dir = -1;
18.
19.        row += dir;
20.      }
21.
22.      StringBuilder res = new StringBuilder();
23.      for (StringBuilder sb : list) {
24.        res.append(sb.toString());
25.      }
26.      return res.toString();
27.    }
28.  }
```

Topological Sort

- Topological Sort

Topological Sort

Alien Dictionary

- Alien Dictionary

Alien Dictionary

There is a new alien language which uses the latin alphabet. However, the order among letters are unknown to you. You receive a list of words from the dictionary, where words are sorted lexicographically by the rules of this new language. Derive the order of letters in this language.

For example,

Given the following words in dictionary,

```
1. [
2.    "wrt",
3.    "wrf",
4.    "er",
5.    "ett",
6.    "rftt"
7. ]
```

The correct order is: `"wertf"` .

Note:

1. You may assume all letters are in lowercase.
2. If the order is invalid, return an empty string.
3. There may be multiple valid order of letters, return any one of them is fine.

Solution:

```
1.  public class Solution {
2.    public String alienOrder(String[] words) {
3.      if (words == null || words.length == 0)
4.        return null;
5.
6.      if (words.length == 1) {
7.        return words[0];
8.      }
9.
10.     Map<Character, List<Character>> adjList = new
    HashMap<Character, List<Character>>();
11.
12.     for (int i = 0; i < words.length - 1; i++) {
13.       String w1 = words[i], w2 = words[i + 1];
14.       int n1 = w1.length(), n2 = w2.length();
15.
16.       boolean found = false;
17.
18.       for (int j = 0; j < Math.max(w1.length(), w2.length()); j++)
    {
19.         Character c1 = j < n1 ? w1.charAt(j) : null, c2 = j < n2 ?
    w2.charAt(j) : null;
20.
21.         if (c1 != null && !adjList.containsKey(c1)) {
22.           adjList.put(c1, new ArrayList<Character>());
23.         }
24.
25.         if (c2 != null && !adjList.containsKey(c2)) {
26.           adjList.put(c2, new ArrayList<Character>());
27.         }
28.
29.         if (c1 != null && c2 != null && c1 != c2 && !found) {
30.           adjList.get(c1).add(c2);
31.           found = true;
32.         }
33.       }
```

```
34.        }
35.
36.      Set<Character> visited = new HashSet<>();
37.      Set<Character> loop = new HashSet<>();
38.      Stack<Character> stack = new Stack<Character>();
39.
40.      for (Character key : adjList.keySet()) {
41.        if (!visited.contains(key)) {
42.          if (!topologicalSort(adjList, key, visited, loop, stack)) {
43.            return "";
44.          }
45.        }
46.      }
47.
48.      StringBuilder sb = new StringBuilder();
49.
50.      while (!stack.isEmpty()) {
51.        sb.append(stack.pop());
52.      }
53.
54.      return sb.toString();
55.    }
56.
57.    boolean topologicalSort(Map<Character, List<Character>> adjList,
      char u, Set<Character> visited, Set<Character> loop,
      Stack<Character> stack) {
58.      visited.add(u);
59.      loop.add(u);
60.
61.      if (adjList.containsKey(u)) {
62.        for (int i = 0; i < adjList.get(u).size(); i++) {
63.          char v = adjList.get(u).get(i);
64.
65.          if (loop.contains(v))
66.            return false;
67.
68.          if (!visited.contains(v)) {
69.            if (!topologicalSort(adjList, v, visited, loop, stack)) {
```

```
70.                  return false;
71.              }
72.          }
73.       }
74.     }
75.
76.    loop.remove(u);
77.
78.    stack.push(u);
79.    return true;
80.   }
81. }
```

Course Schedule

Course Schedule

There are a total of n courses you have to take, labeled from `0` to `n - 1` .

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

For example:

```
1.  2, [[1,0]]
```

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So it is possible.

```
1.  2, [[1,0],[0,1]]
```

There are a total of 2 courses to take. To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

Note:

The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.

Hints:

1. This problem is equivalent to finding if a cycle exists in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via BFS.

Solution:

```
public class Solution {
  public boolean canFinish(int numCourses, int[][] prerequisites) {
    List<List<Integer>> adjList = new ArrayList<List<Integer>>(numCourses);

    for (int i = 0; i < numCourses; i++)
      adjList.add(i, new ArrayList<Integer>());

    for (int i = 0; i < prerequisites.length; i++)
      adjList.get(prerequisites[i][0]).add(prerequisites[i][1]);

    boolean[] visited = new boolean[numCourses];

    for (int u = 0; u < numCourses; u++)
      if (hasCycle(adjList, u, visited, new boolean[numCourses]))
        return false;
```

```
16.
17.     return true;
18.   }
19.
20.   boolean hasCycle(List<List<Integer>> adjList, int u, boolean[]
    visited, boolean[] stack) {
21.     if (visited[u])
22.       return false;
23.
24.     if (stack[u])
25.       return true;
26.
27.     stack[u] = true;
28.
29.     for (Integer v : adjList.get(u))
30.       if (hasCycle(adjList, v, visited, stack))
31.         return true;
32.
33.     visited[u] = true;
34.
35.     return false;
36.   }
37. }
```

Course Schedule II

Course Schedule II

There are a total of n courses you have to take, labeled from `0` to `n - 1` .

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite pairs, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

For example:

```
1.  2, [[1,0]]
```

There are a total of 2 courses to take. To take course 1 you should have finished course 0. So the correct course order is `[0,1]`

```
1.  4, [[1,0],[2,0],[3,1],[3,2]]
```

There are a total of 4 courses to take. To take

course 3 you should have finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0. So one correct course order is `[0,1,2,3]` . Another correct ordering is `[0,2,1,3]` .

Note:

The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.

Hints:

1. This problem is equivalent to finding the topological order in a directed graph. If a cycle exists, no topological ordering exists and therefore it will be impossible to take all courses.
2. Topological Sort via DFS - A great video tutorial (21 minutes) on Coursera explaining the basic concepts of Topological Sort.
3. Topological sort could also be done via BFS.

Solution:

```
1.  public class Solution {
2.    public int[] findOrder(int numCourses, int[][] prerequisites) {
3.      // Use adjacency list
4.      List<List<Integer>> adj = new ArrayList<List<Integer>>
    (numCourses);
5.
6.      for (int i = 0; i < numCourses; i++)
7.        adj.add(i, new ArrayList<>());
```

```
 8.
 9.       for (int i = 0; i < prerequisites.length; i++)
10.         adj.get(prerequisites[i][1]).add(prerequisites[i][0]);
11.
12.       boolean[] visited = new boolean[numCourses];
13.
14.       // reverse result of dfs is the answer
15.       Stack<Integer> stack = new Stack<>();
16.
17.       for (int u = 0; u < numCourses; u++) {
18.         if (!topologicalSort(adj, u, stack, visited, new
    boolean[numCourses]))
19.           return new int[0];
20.       }
21.
22.       int i = 0;
23.       int[] result = new int[numCourses];
24.       while (!stack.isEmpty()) {
25.         result[i++] = stack.pop();
26.       }
27.       return result;
28.     }
29.
30.     private boolean topologicalSort(List<List<Integer>> adj, int u,
    Stack<Integer> stack, boolean[] visited, boolean[] isLoop) {
31.       if (visited[u])
32.         return true;
33.
34.       if (isLoop[u])
35.         return false;
36.
37.       isLoop[u] = true;
38.
39.       for (Integer v : adj.get(u)) {
40.         if (!topologicalSort(adj, v, stack, visited, isLoop))
41.           return false;
42.       }
43.
```

```
44.     visited[u] = true;
45.
46.     stack.push(u);
47.     return true;
48.   }
49. }
```

Longest Increasing Path in a Matrix

- Longest Increasing Path in a Matrix

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 (i.e. wrap-around is not allowed).

Example 1:

```
1.  nums = [
2.    [9,9,4],
3.    [6,6,8],
4.    [2,1,1]
5.  ]
```

Return `4`

The longest increasing path is `[1, 2, 6, 9]` .

Example 2:

```
1.  nums = [
2.    [3,4,5],
3.    [3,2,6],
4.    [2,2,1]
5.  ]
```

Return `4`

The longest increasing path is `[3, 4, 5, 6]` . Moving diagonally is not allowed.

Solution:

```
1.
2.
```

Tree

- Tree

Tree

Balanced Binary Tree

- Balanced Binary Tree

Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

Solution:

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public boolean isBalanced(TreeNode root) {
    return getHeight(root) >= 0;
  }

  int getHeight(TreeNode root) {
    if (root == null)
      return 0;

    int left = getHeight(root.left);
```

```
20.
21.     if (left < 0)
22.       return -1;
23.
24.     int right = getHeight(root.right);
25.
26.     if (right < 0)
27.       return -1;
28.
29.     if (Math.abs(left - right) > 1)
30.       return -1;
31.
32.     return 1 + Math.max(left, right);
33.   }
34. }
```

Binary Search Tree Iterator

- Binary Search Tree Iterator

Binary Search Tree Iterator

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling `next()` will return the next smallest number in the BST.

Note: `next()` and `hasNext()` should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

Solution:

```
 1.  /**
 2.   * Definition for binary tree
 3.   * public class TreeNode {
 4.   *     int val;
 5.   *     TreeNode left;
 6.   *     TreeNode right;
 7.   *     TreeNode(int x) { val = x; }
 8.   * }
 9.   */
10.
11.  public class BSTIterator {
12.    Stack<TreeNode> stack = null;
13.    TreeNode current = null;
14.
15.    public BSTIterator(TreeNode root) {
```

```
16.        current = root;
17.        stack = new Stack<TreeNode>();
18.    }
19.
20.    /** @return whether we have a next smallest number */
21.    public boolean hasNext() {
22.        return !stack.isEmpty() || current != null;
23.    }
24.
25.    /** @return the next smallest number */
26.    public int next() {
27.        while (current != null) {
28.            stack.push(current);
29.            current = current.left;
30.        }
31.
32.        TreeNode t = stack.pop();
33.        current = t.right;
34.        return t.val;
35.    }
36. }
37.
38. /**
39.  * Your BSTIterator will be called like this:
40.  * BSTIterator i = new BSTIterator(root);
41.  * while (i.hasNext()) v[f()] = i.next();
42.  */
```

Binary Tree Inorder Traversal

Binary Tree Inorder Traversal

Given a binary tree, return the inorder traversal of its nodes' values.

For example:

Given binary tree `[1,null,2,3]` ,

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

return `[1,3,2]` .

Note: Recursive solution is trivial, could you do it iteratively?

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
9.   */
```

```
10.  public class Solution {
11.    public List<Integer> inorderTraversal(TreeNode root) {
12.      List<Integer> list = new ArrayList<>();
13.
14.      TreeNode curr = root;
15.      Stack<TreeNode> stack = new Stack<>();
16.
17.      while (!stack.isEmpty() || curr != null) {
18.        if (curr != null) {
19.          stack.push(curr);
20.          curr = curr.left;
21.        } else {
22.          curr = stack.pop();
23.          list.add(curr.val);
24.          curr = curr.right;
25.        }
26.      }
27.
28.      return list;
29.    }
30.  }
```

Binary Tree Level Order Traversal II

- Binary Tree Level Order Traversal II

Binary Tree Level Order Traversal II

Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example:

Given binary tree `[3,9,20,null,null,15,7]` ,

```
1.      3
2.     / \
3.    9  20
4.      /  \
5.    15   7
```

return its bottom-up level order traversal as:

```
1. [
2.   [15,7],
3.   [9,20],
4.   [3]
5. ]
```

Solution:

```
1. /**
2.  * Definition for binary tree
3.  * public class TreeNode {
4.  *     int val;
5.  *     TreeNode left;
```

```
 6.    *       TreeNode right;
 7.    *       TreeNode(int x) { val = x; }
 8.    * }
 9.    */
10.   public class Solution {
11.     public List<List<Integer>> levelOrderBottom(TreeNode root) {
12.       List<List<Integer>> res = new ArrayList<List<Integer>>();
13.       List<Integer> list = new ArrayList<Integer>();
14.
15.       if (root == null) {
16.         return res;
17.       }
18.
19.       Queue<TreeNode> curr = new LinkedList<TreeNode>();
20.       Queue<TreeNode> next = new LinkedList<TreeNode>();
21.
22.       curr.add(root);
23.
24.       while (!curr.isEmpty()) {
25.         TreeNode node = curr.poll();
26.         list.add(node.val);
27.
28.         if (node.left != null) {
29.           next.add(node.left);
30.         }
31.
32.         if (node.right != null) {
33.           next.add(node.right);
34.         }
35.
36.         if (curr.isEmpty()) {
37.           res.add(0, list);
38.           list = new ArrayList<Integer>();
39.
40.           curr = next;
41.           next = new LinkedList<TreeNode>();
42.         }
43.       }
```

```
44.
45.        return res;
46.    }
47. }
```

Binary Tree Level Order Traversal

- Binary Tree Level Order Traversal

Binary Tree Level Order Traversal

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example:

Given binary tree `[3,9,20,null,null,15,7]` ,

```
1.      3
2.     / \
3.    9  20
4.      /  \
5.     15   7
```

return its level order traversal as:

```
1.  [
2.    [3],
3.    [9,20],
4.    [15,7]
5.  ]
```

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
```

```
 6.   *      TreeNode right;
 7.   *      TreeNode(int x) { val = x; }
 8.   * }
 9.   */
10.  public class Solution {
11.    public List<List<Integer>> levelOrder(TreeNode root) {
12.      List<List<Integer>> res = new ArrayList<List<Integer>>();
13.
14.      if (root == null) {
15.        return res;
16.      }
17.
18.      // Use a queue to help level order traversal
19.      Queue<TreeNode> queue = new LinkedList<TreeNode>();
20.
21.      queue.offer(root);
22.      queue.offer(null);
23.
24.      List<Integer> list = new ArrayList<Integer>();
25.
26.      while (!queue.isEmpty()) {
27.        TreeNode node = queue.poll();
28.
29.        if (node != null) {
30.          list.add(node.val);
31.
32.          if (node.left != null) {
33.            queue.offer(node.left);
34.          }
35.
36.          if (node.right != null) {
37.            queue.offer(node.right);
38.          }
39.        } else {
40.          res.add(new ArrayList<Integer>(list));
41.          list = new ArrayList<Integer>();
42.
43.          if (!queue.isEmpty()) {
```

```
44.              queue.offer(null);
45.          }
46.        }
47.      }
48.
49.      return res;
50.    }
51.  }
```

Binary Tree Longest Consecutive Sequence

- Binary Tree Longest Consecutive Sequence

Binary Tree Longest Consecutive Sequence

Given a binary tree, find the length of the longest consecutive sequence path.

The path refers to any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The longest consecutive path need to be from parent to child (cannot be the reverse).

For example,

```
1.     1
2.      \
3.       3
4.      / \
5.     2   4
6.          \
7.           5
```

Longest consecutive sequence path is `3-4-5`, so return `3`.

```
1.     2
2.      \
3.       3
4.      /
```

```
5.    2
6.   /
7.  1
```

Longest consecutive sequence path is `2-3` ,not `3-2-1` , so return `2` .

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
9.   */
10. public class Solution {
11.   int max;
12.
13.   public int longestConsecutive(TreeNode root) {
14.     max = 0;
15.     helper(root);
16.     return max;
17.   }
18.
19.   int helper(TreeNode root) {
20.     if (root == null) return 0;
21.
22.     int len = 1;
23.     int left = helper(root.left);
24.     int right = helper(root.right);
25.
26.     if (root.left != null && root.val == root.left.val - 1) {
27.       len = Math.max(len, left + 1);
28.     }
```

```
29.
30.     if (root.right != null && root.val == root.right.val - 1) {
31.       len = Math.max(len, right + 1);
32.     }
33.
34.     max = Math.max(len, max);
35.
36.     return len;
37.   }
38. }
```

Binary Tree Maximum Path Sum

- Binary Tree Maximum Path Sum

Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

For this problem, a path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path does not need to go through the root.

For example:
Given the below binary tree,

```
1.       1
2.      / \
3.     2   3
```

Return 6 .

Solution:

```
 1.  /**
 2.   * Definition for a binary tree node.
 3.   * public class TreeNode {
 4.   *     int val;
 5.   *     TreeNode left;
 6.   *     TreeNode right;
 7.   *     TreeNode(int x) { val = x; }
 8.   * }
 9.   */
10.  public class Solution {
```

```
11.      int max = Integer.MIN_VALUE;
12.
13.      public int maxPathSum(TreeNode root) {
14.        helper(root);
15.        return max;
16.      }
17.
18.      int helper(TreeNode root) {
19.        if (root == null) return 0;
20.
21.        int left = Math.max(helper(root.left), 0);
22.        int right = Math.max(helper(root.right), 0);
23.
24.        max = Math.max(max, root.val + left + right);
25.
26.        return root.val + Math.max(left, right);
27.      }
28.  }
```

Binary Tree Paths

- Binary Tree Paths

Binary Tree Paths

Given a binary tree, return all root-to-leaf paths.

For example, given the following binary tree:

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

All root-to-leaf paths are:

```
1.  ["1->2->5", "1->3"]
```

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
9.   */
10. public class Solution {
11.   public List<String> binaryTreePaths(TreeNode root) {
12.     List<String> res = new ArrayList<String>();
13.     helper(root, new ArrayList<Integer>(), res);
```

```
14.      return res;
15.    }
16.
17.    void helper(TreeNode root, List<Integer> sol, List<String> res) {
18.      if (root == null) return;
19.
20.      sol.add(root.val);
21.
22.      if (root.left == null && root.right == null) {
23.        res.add(convert(sol));
24.      } else {
25.        helper(root.left, sol, res);
26.        helper(root.right, sol, res);
27.      }
28.
29.      sol.remove(sol.size() - 1);
30.    }
31.
32.    String convert(List<Integer> list) {
33.      StringBuilder sb = new StringBuilder();
34.
35.      for (int i = 0; i < list.size(); i++) {
36.        if (i > 0) sb.append("->");
37.        sb.append(list.get(i));
38.      }
39.
40.      return sb.toString();
41.    }
42.  }
```

Binary Tree Postorder Traversal

- Binary Tree Postorder Traversal

Binary Tree Postorder Traversal

Given a binary tree, return the postorder traversal of its nodes' values.

For example:

Given binary tree `{1,#,2,3}`,

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

return `[3,2,1]`.

Note: Recursive solution is trivial, could you do it iteratively?

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
9.   */
```

```
10.  public class Solution {
11.    public List<Integer> postorderTraversal(TreeNode root) {
12.      List<Integer> res = new ArrayList<Integer>();
13.
14.      if (root == null)
15.        return res;
16.
17.      Stack<TreeNode> s1 = new Stack<TreeNode>();
18.      Stack<TreeNode> s2 = new Stack<TreeNode>();
19.
20.      s1.push(root);
21.
22.      while (!s1.isEmpty()) {
23.        TreeNode node = s1.pop();
24.        s2.push(node);
25.
26.        if (node.left != null)
27.          s1.push(node.left);
28.
29.        if (node.right != null)
30.          s1.push(node.right);
31.      }
32.
33.      while (!s2.isEmpty())
34.        res.add(s2.pop().val);
35.
36.      return res;
37.    }
38.  }
```

Binary Tree Preorder Traversal

- Binary Tree Preorder Traversal

Binary Tree Preorder Traversal

Given a binary tree, return the preorder traversal of its nodes' values.

For example:

Given binary tree `{1,#,2,3}` ,

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

return `[1,2,3]` .

Note: Recursive solution is trivial, could you do it iteratively?

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
9.   */
```

```
10.  public class Solution {
11.    public List<Integer> preorderTraversal(TreeNode root) {
12.      List<Integer> res = new ArrayList<Integer>();
13.
14.      if (root == null)
15.        return res;
16.
17.      Stack<TreeNode> stack = new Stack<TreeNode>();
18.      stack.push(root);
19.
20.      while (!stack.isEmpty()) {
21.        TreeNode node = stack.pop();
22.        res.add(node.val);
23.
24.        if (node.right != null)
25.          stack.push(node.right);
26.
27.        if (node.left != null)
28.          stack.push(node.left);
29.      }
30.
31.      return res;
32.    }
33.  }
```

Binary Tree Right Side View

- Binary Tree Right Side View

Binary Tree Right Side View

Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

For example:

Given the following binary tree,

```
1.    1           <---
2.   /  \
3.  2    3         <---
4.   \    \
5.    5    4       <---
```

You should return `[1, 3, 4]` .

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
9.   */
10. public class Solution {
11.   public List<Integer> rightSideView(TreeNode root) {
```

```
12.        List<Integer> res = new ArrayList<Integer>();
13.
14.        if (root == null)
15.          return res;
16.
17.        Queue<TreeNode> queue = new LinkedList<TreeNode>();
18.        queue.add(root);
19.        queue.add(null);
20.
21.        List<Integer> list = new ArrayList<Integer>();
22.
23.        while (!queue.isEmpty()) {
24.          TreeNode node = queue.poll();
25.
26.          if (node != null) {
27.            list.add(node.val);
28.
29.            if (node.left != null)
30.              queue.add(node.left);
31.
32.            if (node.right != null)
33.              queue.add(node.right);
34.          } else {
35.            res.add(list.get(list.size() - 1));
36.
37.            if (!queue.isEmpty())
38.              queue.add(null);
39.          }
40.        }
41.
42.        return res;
43.    }
44.  }
```

Binary Tree Upside Down

- Binary Tree Upside Down

Binary Tree Upside Down

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

For example:

Given a binary tree `{1,2,3,4,5}` ,

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

return the root of the binary tree `[4,5,2,#,#,3,1]` .

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

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
9.   */
10. public class Solution {
11.   public TreeNode upsideDownBinaryTree(TreeNode root) {
12.     if (root == null || root.left == null && root.right == null)
13.       return root;
14.
15.     TreeNode newRoot = upsideDownBinaryTree(root.left);
16.
17.     root.left.left = root.right;
18.     root.left.right = root;
19.
20.     root.left = null;
21.     root.right = null;
22.
23.     return newRoot;
24.   }
25. }
```

Binary Tree Zigzag Level Order Traversal

Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:

Given binary tree `[3,9,20,null,null,15,7]`,

```
1.      3
2.     / \
3.    9  20
4.      /  \
5.     15   7
```

return its zigzag level order traversal as:

```
1.  [
2.    [3],
3.    [20,9],
4.    [15,7]
5.  ]
```

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
```

```
 3.   * public class TreeNode {
 4.   *     int val;
 5.   *     TreeNode left;
 6.   *     TreeNode right;
 7.   *     TreeNode(int x) { val = x; }
 8.   * }
 9.   */
10.  public class Solution {
11.    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
12.      List<List<Integer>> res = new ArrayList<>();
13.
14.      if (root == null) {
15.        return res;
16.      }
17.
18.      List<Integer> list = new ArrayList<>();
19.
20.      Stack<TreeNode> s1 = new Stack<>();
21.      Stack<TreeNode> s2 = new Stack<>();
22.
23.      boolean leftToRight = true;
24.
25.      s1.push(root);
26.
27.      while (!s1.isEmpty()) {
28.        TreeNode node = s1.pop();
29.
30.        list.add(node.val);
31.
32.        if (leftToRight) {
33.          if (node.left != null) {
34.            s2.push(node.left);
35.          }
36.
37.          if (node.right != null) {
38.            s2.push(node.right);
39.          }
40.        } else {
```

```
41.          if (node.right != null) {
42.            s2.push(node.right);
43.          }
44.
45.          if (node.left != null) {
46.            s2.push(node.left);
47.          }
48.        }
49.
50.        if (s1.isEmpty()) {
51.          leftToRight = !leftToRight;
52.          res.add(new ArrayList<>(list));
53.          list = new ArrayList<>();
54.
55.          s1 = s2;
56.          s2 = new Stack<>();
57.        }
58.      }
59.
60.      return res;
61.    }
62.  }
```

Closest Binary Search Tree Value

-

Closest Binary Search Tree Value

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

Note:

- Given target value is a floating point.
- You are guaranteed to have only one unique value in the BST that is closest to the target.

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public int closestValue(TreeNode root, double target) {
    int closest = root.val;

    while (root != null) {
      if (Math.abs(root.val - target) < Math.abs(closest - target))
  {
        closest = root.val;
```

```
17.          }
18.
19.          if (target < root.val) {
20.              root = root.left;
21.          } else {
22.              root = root.right;
23.          }
24.      }
25.
26.      return closest;
27.  }
28. }
```

Closest Binary Search Tree Value II

- Closest Binary Search Tree Value II

Closest Binary Search Tree Value II

Given a non-empty binary search tree and a target value, find k values in the BST that are closest to the target.

Note:

- Given target value is a floating point.
- You may assume k is always valid, that is: k ≤ total nodes.
- You are guaranteed to have only one unique set of k values in the BST that are closest to the target.

Follow up:

Assume that the BST is balanced, could you solve it in less than O(n) runtime (where n = total nodes)?

Hint:

- Consider implement these two helper functions:
  i. getPredecessor(N), which returns the next smaller node to N.
  ii. getSuccessor(N), which returns the next larger node to N.
- Try to assume that each node has a parent

pointer, it makes the problem much easier.

- Without parent pointer we just need to keep track of the path from the root to the current node using a stack.
- You would need two stacks to track the path in finding predecessor and successor node separately.

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public List<Integer> closestKValues(TreeNode root, double target,
int k) {
    List<Integer> res = new ArrayList<>();
    Stack<Integer> s1 = inorder(root, target, true);  //
predecessors
    Stack<Integer> s2 = inorder(root, target, false); // successors

    while (k-- > 0) {
      if (s1.isEmpty())
        res.add(s2.pop());
      else if (s2.isEmpty())
        res.add(s1.pop());
      else if (Math.abs(s1.peek() - target) < Math.abs(s2.peek() -
target))
        res.add(s1.pop());
      else
```

```
24.          res.add(s2.pop());
25.      }
26.
27.      return res;
28.    }
29.
30.    // iterative inorder traversal
31.    Stack<Integer> inorder(TreeNode root, double target, boolean
    reverse) {
32.      Stack<Integer> res = new Stack<>();
33.      Stack<TreeNode> stack = new Stack<>();
34.      TreeNode curr = root;
35.
36.      while (!stack.isEmpty() || curr != null) {
37.        if (curr != null) {
38.          stack.push(curr);
39.          curr = reverse ? curr.right : curr.left;
40.        } else {
41.          curr = stack.pop();
42.          if (reverse && curr.val <= target) break;
43.          if (!reverse && curr.val > target) break;
44.          res.push(curr.val);
45.          curr = reverse ? curr.left : curr.right;
46.        }
47.      }
48.      return res;
49.    }
50. }
```

Construct Binary Tree from Inorder and Postorder Traversal

- Construct Binary Tree from Inorder and Postorder Traversal

Construct Binary Tree from Inorder and Postorder Traversal

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

Note:

You may assume that duplicates do not exist in the tree.

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public TreeNode buildTree(int[] inorder, int[] postorder) {
    Map<Integer, Integer> map = new HashMap<Integer, Integer>();

    for (int i = 0; i < inorder.length; i++) {
      map.put(inorder[i], i);
    }
```

```java
17.
18.      return build(postorder, 0, postorder.length - 1, inorder, 0,
    inorder.length - 1, map);
19.    }
20.
21.    TreeNode build(int[] post, int i1, int j1, int[] in, int i2, int
    j2, Map<Integer, Integer> map) {
22.      if (i1 > j1 || i2 > j2) {
23.        return null;
24.      }
25.
26.      TreeNode root = new TreeNode(post[j1]);
27.
28.      int index = map.get(root.val);
29.      int left = index - i2;
30.
31.      root.left = build(post, i1, i1 + left - 1, in, i2, index - 1,
    map);
32.      root.right = build(post, i1 + left, j1 - 1, in, index + 1, j2,
    map);
33.
34.      return root;
35.    }
36. }
```

Construct Binary Tree from Preorder and Inorder Traversal

- Construct Binary Tree from Preorder and Inorder Traversal

Construct Binary Tree from Preorder and Inorder Traversal

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

Note:

You may assume that duplicates do not exist in the tree.

Solution:

```
 1.  /**
 2.   * Definition for a binary tree node.
 3.   * public class TreeNode {
 4.   *     int val;
 5.   *     TreeNode left;
 6.   *     TreeNode right;
 7.   *     TreeNode(int x) { val = x; }
 8.   * }
 9.   */
10.  public class Solution {
11.    public TreeNode buildTree(int[] preorder, int[] inorder) {
12.      Map<Integer, Integer> map = new HashMap<>();
13.
14.      for (int i = 0; i < inorder.length; i++) {
15.        map.put(inorder[i], i);
16.      }
```

```
17.
18.     return build(preorder, 0, preorder.length - 1, inorder, 0,
    inorder.length - 1, map);
19.   }
20.
21.   TreeNode build(int[] pre, int i1, int j1, int[] in, int i2, int
    j2, Map<Integer, Integer> map) {
22.     if (i1 > j1 || i2 > j2) {
23.       return null;
24.     }
25.
26.     TreeNode root = new TreeNode(pre[i1]);
27.
28.     int index = map.get(root.val);
29.     int nleft = index - i2;
30.
31.     root.left = build(pre, i1 + 1, i1 + nleft, in, i2, index - 1,
    map);
32.     root.right = build(pre, i1 + nleft + 1, j1, in, index + 1, j2,
    map);
33.
34.     return root;
35.   }
36. }
```

Convert Sorted Array to Binary Search Tree

- Convert Sorted Array to Binary Search Tree

Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public TreeNode sortedArrayToBST(int[] nums) {
    return build(nums, 0, nums.length - 1);
  }

  TreeNode build(int[] nums, int lo, int hi) {
    if (lo > hi)
      return null;

    int mid = lo + (hi - lo) / 2;

    TreeNode root = new TreeNode(nums[mid]);

```

```
23.        root.left = build(nums, lo, mid - 1);
24.        root.right = build(nums, mid + 1, hi);
25.
26.        return root;
27.    }
28. }
```

Count Complete Tree Nodes

- Count Complete Tree Nodes

Count Complete Tree Nodes

Given a complete binary tree, count the number of nodes.

Definition of a complete binary tree from Wikipedia:

In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes inclusive at the last level h.

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public int countNodes(TreeNode root) {
    if (root == null)
      return 0;

    int hLeft = getHeight(root.left);
```

```
16.        int hRight = getHeight(root.right);
17.
18.        if (hLeft == hRight)
19.            return (1 << hLeft) + countNodes(root.right);
20.        else
21.            return (1 << hRight) + countNodes(root.left);
22.    }
23.
24.    int getHeight(TreeNode root) {
25.        if (root == null)
26.            return 0;
27.
28.        return 1 + getHeight(root.left);
29.    }
30. }
```

Count Univalue Subtrees

-
-

Count Univalue Subtrees

Given a binary tree, count the number of uni-value subtrees.

A Uni-value subtree means all nodes of the subtree have the same value.

For example:

Given binary tree,

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

return 4 .

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
```

```
 9.    */
10.  public class Solution {
11.    int count;
12.
13.    public int countUnivalSubtrees(TreeNode root) {
14.      count = 0;
15.      helper(root);
16.      return count;
17.    }
18.
19.    boolean helper(TreeNode root) {
20.      if (root == null)
21.        return true;
22.
23.      boolean left = helper(root.left);
24.      boolean right = helper(root.right);
25.
26.      if (left && right && (root.left == null || root.val ==
    root.left.val) && (root.right == null || root.val ==
    root.right.val)) {
27.        count++;
28.        return true;
29.      }
30.
31.      return false;
32.    }
33.  }
```

Check Univalue Tree

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
```

```
 6.   *      TreeNode right;
 7.   *      TreeNode(int x) { val = x; }
 8.   * }
 9.   */
10.  public class Solution {
11.    public boolean isUnivalTree(TreeNode root) {
12.      if (root == null)
13.        return false;
14.
15.      return helper(root, root.val);
16.    }
17.
18.    boolean helper(TreeNode root, in val) {
19.      if (root == null)
20.        return true;
21.
22.      if (root.val != val)
23.        return false;
24.
25.      return helper(root.left, val) && helper(root.right, val);
26.    }
27.  }
```

Find Leaves of Binary Tree

- Find Leaves of Binary Tree

Find Leaves of Binary Tree

Given a binary tree, collect a tree's nodes as if you were doing this: Collect and remove all leaves, repeat until the tree is empty.

Example:

Given binary tree

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

Returns `[4, 5, 3], [2], [1]` .

Explanation:

1. Removing the leaves `[4, 5, 3]` would result in this tree:

   ```
   1.          1
   2.         /
   3.        2
   ```

2. Now removing the leaf `[2]` would result in this tree:

   ```
   1.          1
   ```

3. Now removing the leaf `[1]` would result in the empty tree:

```
1.          []
```

Returns `[4, 5, 3], [2], [1]` .

Solution:

```
1.
2.
```

Flatten Binary Tree to Linked List

-

Flatten Binary Tree to Linked List

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

For example,

Given

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

The flattened tree should look like:

```
 1.      1
 2.       \
 3.        2
 4.         \
 5.          3
 6.           \
 7.            4
 8.             \
 9.              5
10.               \
11.                6
```

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
9.   */
10. public class Solution {
11.   public void flatten(TreeNode root) {
12.     if (root == null)
13.       return;
14.
15.     Stack<TreeNode> stack = new Stack<>();
16.     stack.push(root);
17.
18.     TreeNode last = null;
19.
20.     while (!stack.isEmpty()) {
21.       TreeNode curr = stack.pop();
22.
23.       if (last != null) {
24.         last.left = null;
25.         last.right = curr;
26.       }
27.
28.       if (curr.right != null)
29.         stack.push(curr.right);
30.
31.       if (curr.left != null)
32.         stack.push(curr.left);
33.
34.       last = curr;
35.     }
36.   }
37. }
```

House Robber III

House Robber III

The thief has found himself a new place for his thievery again. There is only one entrance to this area, called the "root." Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that "all houses in this place forms a binary tree". It will automatically contact the police if two directly-linked houses were broken into on the same night.

Determine the maximum amount of money the thief can rob tonight without alerting the police.

Example 1:

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

Maximum amount of money the thief can `rob = 3 + 3 + 1 =` `7` .

Example 2:

```
1.      3
2.     / \
```

```
3.     4   5
4.    / \   \
5.   1   3   1
```

Maximum amount of money the thief can `rob = 4 + 5 = 9` .

Solution:

```
1.
2.
```

Inorder Successor in BST

- Inorder Successor in BST

Inorder Successor in BST

Given a binary search tree and a node in it, find the in-order successor of that node in the BST.

Note: If the given node has no in-order successor in the tree, return `null` .

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {
    if (p.right != null) {
      return getMin(p.right);
    }

    TreeNode succ = null;

    while (root != null) {
      if (root.val <= p.val) {
        root = root.right;
      } else {
        succ = root;
```

```
23.          root = root.left;
24.        }
25.      }
26.
27.      return succ;
28.    }
29.
30.    TreeNode getMin(TreeNode p) {
31.      while (p.left != null) {
32.        p = p.left;
33.      }
34.      return p;
35.    }
36.  }
```

Invert Binary Tree

Invert Binary Tree

Invert a binary tree.

```
1.      4
2.    /   \
3.   2     7
4.  / \   / \
5. 1   3 6   9
```

to

```
1.      4
2.    /   \
3.   7     2
4.  / \   / \
5. 9   6 3   1
```

Trivia:

This problem was inspired by this original tweet by Max Howell:

> Google: 90% of our engineers use the software you wrote (Homebrew),
> but you can't invert a binary tree on a whiteboard so fuck off.

Solution:

```
1. /**
2.  * Definition for a binary tree node.
3.  * public class TreeNode {
```

```
 4.  *      int val;
 5.  *      TreeNode left;
 6.  *      TreeNode right;
 7.  *      TreeNode(int x) { val = x; }
 8.  * }
 9.  */
10.  public class Solution {
11.    public TreeNode invertTree(TreeNode root) {
12.      if (root == null)
13.        return null;
14.
15.      invertTree(root.left);
16.      invertTree(root.right);
17.
18.      TreeNode tmp = root.left;
19.      root.left = root.right;
20.      root.right = tmp;
21.
22.      return root;
23.    }
24.  }
```

Kth Smallest Element in a BST

- Kth Smallest Element in a BST

Kth Smallest Element in a BST

Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.

Note:

You may assume k is always valid, 1 ≤ k ≤ BST's total elements.

Follow up:

What if the BST is modified (insert/delete operations) often and you need to find the kth smallest frequently? How would you optimize the kthSmallest routine?

Hint:

1. Try to utilize the property of a BST.
2. What if you could modify the BST node's structure?
3. The optimal runtime complexity is O(height of BST).

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
```

```
 3.    * public class TreeNode {
 4.    *     int val;
 5.    *     TreeNode left;
 6.    *     TreeNode right;
 7.    *     TreeNode(int x) { val = x; }
 8.    * }
 9.    */
10.  public class Solution {
11.    public int kthSmallest(TreeNode root, int k) {
12.      int nLeft = countNodes(root.left);
13.
14.      if (nLeft == k - 1)
15.        return root.val;
16.
17.      else if (nLeft > k - 1)
18.        return kthSmallest(root.left, k);
19.
20.      else
21.        return kthSmallest(root.right, k - nLeft - 1);
22.    }
23.
24.    int countNodes(TreeNode root) {
25.      if (root == null)
26.        return 0;
27.
28.      return 1 + countNodes(root.left) + countNodes(root.right);
29.    }
30.  }
```

Largest BST Subtree

- Largest BST Subtree

Largest BST Subtree

Given a binary tree, find the largest subtree which is a Binary Search Tree (BST), where largest means subtree with largest number of nodes in it.

Note:

A subtree must include all of its descendants.

Here's an example:

```
1.      10
2.     / \
3.    5  15
4.   / \   \
5.  1  8   7
```

The Largest BST Subtree in this case is the highlighted one.

The return value is the subtree's size, which is 3.

Hint:

1. You can recursively use algorithm similar to 98. Validate Binary Search Tree at each node of the tree, which will result in O(nlogn) time complexity.

Follow up:

Can you figure out ways to solve it with O(n) time
complexity?

Lowest Common Ancestor of a Binary Search Tree

- Lowest Common Ancestor of a Binary Search Tree

Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself)."

```
1.            _____6_____
2.           /              \
3.        ___2__          ___8__
4.       /      \        /      \
5.      0       _4      7        9
6.             /  \
7.            3    5
```
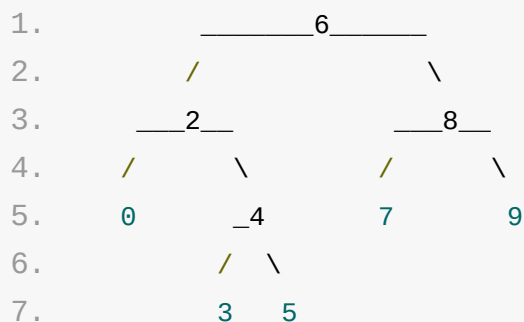
For example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. Another example is LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
```

```
 3.   * public class TreeNode {
 4.   *     int val;
 5.   *     TreeNode left;
 6.   *     TreeNode right;
 7.   *     TreeNode(int x) { val = x; }
 8.   * }
 9.   */
10.  public class Solution {
11.    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
     TreeNode q) {
12.      if (root == null)
13.        return null;
14.
15.      if (root.val > p.val && root.val > q.val)
16.        return lowestCommonAncestor(root.left, p, q);
17.      else if (root.val < p.val && root.val < q.val)
18.        return lowestCommonAncestor(root.right, p, q);
19.      else
20.        return root;
21.    }
22.  }
```

Lowest Common Ancestor of a Binary Tree

-

Lowest Common Ancestor of a Binary Tree

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

According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself)."

```
1.            _____3_____
2.           /               \
3.        ___5__          ___1__
4.       /      \        /      \
5.      6       _2       0       8
6.             /  \
7.            7    4
```
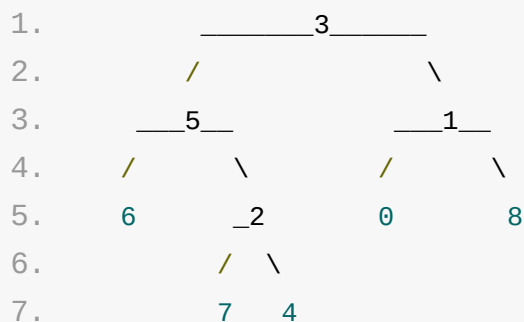
For example, the lowest common ancestor (LCA) of nodes 5 and 1 is 3. Another example is LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

Solution:

```
1. /**
2.  * Definition for a binary tree node.
```

```
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
9.   */
10.  public class Solution {
11.    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
     TreeNode q) {
12.      if (root == null || root == p || root == q)
13.        return root;
14.
15.      TreeNode left = lowestCommonAncestor(root.left, p, q);
16.      TreeNode right = lowestCommonAncestor(root.right, p, q);
17.
18.      if (left != null && right != null)
19.        return root;
20.      else
21.        return left != null ? left : right;
22.    }
23.  }
```

Maximum Depth of Binary Tree

Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Solution:

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public int maxDepth(TreeNode root) {
    if (root == null) {
      return 0;
    }

    return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));
  }
}
```

Minimum Depth of Binary Tree

- Minimum Depth of Binary Tree

Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Solution:

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public int minDepth(TreeNode root) {
    if (root == null)
      return 0;

    if (root.left == null)
      return 1 + minDepth(root.right);

    if (root.right == null)
      return 1 + minDepth(root.left);

    return 1 + Math.min(minDepth(root.left), minDepth(root.right));
  }
```

```
23.  }
```

Path Sum

- Path Sum

Path Sum

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

For example:

Given the below binary tree and `sum = 22`,

```
1.              5
2.             / \
3.            4   8
4.           /   / \
5.         11  13  4
6.        /  \      \
7.       7    2      1
```

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

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
```

```
 9.    */
10.  public class Solution {
11.    public boolean hasPathSum(TreeNode root, int sum) {
12.      if (root == null)
13.        return false;
14.
15.      if (root.left == null && root.right == null && root.val == sum)
16.        return true;
17.
18.      return hasPathSum(root.left, sum - root.val) ||
     hasPathSum(root.right, sum - root.val);
19.    }
20.  }
```

Path Sum II

- Path Sum II

Path Sum II

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example:

Given the below binary tree and `sum = 22` ,

```
1.                5
2.               / \
3.              4   8
4.             /   / \
5.           11  13  4
6.           / \    / \
7.          7   2  5   1
```

return

```
1.  [
2.     [5,4,11,2],
3.     [5,8,4,5]
4.  ]
```

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
```

```
 6.    *       TreeNode right;
 7.    *       TreeNode(int x) { val = x; }
 8.    * }
 9.    */
10.   public class Solution {
11.     public List<List<Integer>> pathSum(TreeNode root, int sum) {
12.       List<List<Integer>> res = new ArrayList<>();
13.       List<Integer> sol = new ArrayList<>();
14.
15.       helper(root, sum, sol, res);
16.
17.       return res;
18.     }
19.
20.     void helper(TreeNode root, int sum, List<Integer> sol,
      List<List<Integer>> res) {
21.       if (root == null)
22.         return;
23.
24.       sol.add(root.val);
25.
26.       if (root.left == null && root.right == null && root.val == sum)
      {
27.         res.add(new ArrayList<>(sol));
28.       } else {
29.         helper(root.left, sum - root.val, sol, res);
30.         helper(root.right, sum - root.val, sol, res);
31.       }
32.
33.       sol.remove(sol.size() - 1);
34.     }
35.   }
```

Populating Next Right Pointers in Each Node

Populating Next Right Pointers in Each Node

Given a binary tree

```
1.    struct TreeLinkNode {
2.      TreeLinkNode *left;
3.      TreeLinkNode *right;
4.      TreeLinkNode *next;
5.    }
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Note:

- You may only use constant extra space.
- You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).

For example,

Given the following perfect binary tree,

```
1.        1
2.      / \
```

```
3.        2    3
4.      / \  / \
5.     4 5  6  7
```

After calling your function, the tree should look like:

```
1.          1 -> NULL
2.        /  \
3.       2 -> 3 -> NULL
4.      / \  / \
5.     4->5->6->7 -> NULL
```

Solution:

```
1.  /**
2.   * Definition for binary tree with next pointer.
3.   * public class TreeLinkNode {
4.   *     int val;
5.   *     TreeLinkNode left, right, next;
6.   *     TreeLinkNode(int x) { val = x; }
7.   * }
8.   */
9.  public class Solution {
10.    public void connect(TreeLinkNode root) {
11.      while (root != null) {
12.        TreeLinkNode curr = root;
13.
14.        while (curr != null) {
15.          if (curr.left != null)
16.            curr.left.next = curr.right;
17.
18.          if (curr.right != null && curr.next != null)
19.            curr.right .next = curr.next.left;
20.
21.          curr = curr.next;
22.        }
```

```
23.
24.        root = root.left;
25.     }
26.   }
27. }
```

Populating Next Right Pointers in Each Node II

Populating Next Right Pointers in Each Node II

Follow up for problem "Populating Next Right Pointers in Each Node".

What if the given tree could be any binary tree? Would your previous solution still work?

Note:

- You may only use constant extra space.

For example,

Given the following binary tree,

```
1.          1
2.         / \
3.        2   3
4.       / \   \
5.      4  5   7
```

After calling your function, the tree should look like:

```
1.          1 -> NULL
2.         / \
3.        2 -> 3 -> NULL
4.       / \   \
```

```
5.        4-> 5 -> 7 -> NULL
```

Solution:

```java
1.  /**
2.   * Definition for binary tree with next pointer.
3.   * public class TreeLinkNode {
4.   *     int val;
5.   *     TreeLinkNode left, right, next;
6.   *     TreeLinkNode(int x) { val = x; }
7.   * }
8.   */
9.  public class Solution {
10.   public void connect(TreeLinkNode root) {
11.     while (root != null) {
12.       TreeLinkNode curr = root;
13.
14.       while (curr != null) {
15.         if (curr.left != null) {
16.           if (curr.right != null) {
17.             curr.left.next = curr.right;
18.           } else {
19.             curr.left.next = getNext(curr.next);
20.           }
21.         }
22.
23.         if (curr.right != null) {
24.           curr.right.next = getNext(curr.next);
25.         }
26.
27.         curr = curr.next;
28.       }
29.
30.       root = getNext(root);
31.     }
32.   }
33.
34.   TreeLinkNode getNext(TreeLinkNode node) {
```

```
35.        while (node != null) {
36.          if (node.left != null)
37.            return node.left;
38.
39.          if (node.right != null)
40.            return node.right;
41.
42.          node = node.next;
43.        }
44.
45.        return null;
46.      }
47.  }
```

Recover Binary Search Tree

- Recover Binary Search Tree

Recover Binary Search Tree

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note:

A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
9.   */
10. public class Solution {
11.   public void recoverTree(TreeNode root) {
12.     if (root == null)
13.       return;
14.
15.     TreeNode a = null;
16.     TreeNode b = null;
17.
18.     TreeNode prev = null;
```

```
19.        TreeNode curr = root;
20.        Stack<TreeNode> stack = new Stack<>();
21.
22.        while (!stack.isEmpty() || curr != null) {
23.          if (curr != null) {
24.            stack.push(curr);
25.            curr = curr.left;
26.          } else {
27.            curr = stack.pop();
28.
29.            if (prev != null && prev.val > curr.val) {
30.              if (a == null) {
31.                a = prev;
32.              }
33.
34.              b = curr;
35.            }
36.
37.            prev = curr;
38.            curr = curr.right;
39.          }
40.        }
41.
42.        // swap a and b to fix the problem
43.        if (a != null && b != null) {
44.          int tmp = a.val;
45.          a.val = b.val;
46.          b.val = tmp;
47.        }
48.      }
49.  }
```

Same Tree

- Same Tree

Same Tree

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

Solution:

```
 1.  /**
 2.   * Definition for a binary tree node.
 3.   * public class TreeNode {
 4.   *     int val;
 5.   *     TreeNode left;
 6.   *     TreeNode right;
 7.   *     TreeNode(int x) { val = x; }
 8.   * }
 9.   */
10.  public class Solution {
11.    public boolean isSameTree(TreeNode p, TreeNode q) {
12.      if (p == null && q == null) {
13.        return true;
14.      }
15.
16.      if (p == null || q == null) {
17.        return false;
18.      }
19.
20.      if (p.val != q.val) {
```

```
21.         return false;
22.     }
23.
24.     return isSameTree(p.left, q.left) && isSameTree(p.right,
    q.right);
25.   }
26. }
```

Serialize and Deserialize Binary Tree

- Serialize and Deserialize Binary Tree

Serialize and Deserialize Binary Tree

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

For example, you may serialize the following tree

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

as `"[1,2,3,null,null,4,5]"` , just the same as how LeetCode OJ serializes a binary tree. You do not necessarily need to follow this format, so please be

creative and come up with different approaches yourself.

Note: Do not use class member/global/static variables to store states. Your serialize and deserialize algorithms should be stateless.

Solution:

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

  // Encodes a tree to a single string.
  public String serialize(TreeNode root) {
    if (root == null) return null;

    String delim = "";
    StringBuilder sb = new StringBuilder();

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

    // preorder traversal
    while (!stack.isEmpty()) {
      TreeNode node = stack.pop();
      sb.append(delim).append(node == null ? "#" :
String.valueOf(node.val));
      delim = ",";

```

```
28.        if (node != null) {
29.          stack.push(node.right);
30.          stack.push(node.left);
31.        }
32.      }
33.
34.      return sb.toString();
35.    }
36.
37.    // Decodes your encoded data to tree.
38.    public TreeNode deserialize(String data) {
39.      if (data == null) return null;
40.
41.      String[] list = data.split(",");
42.
43.      // create the root node and push it to the stack
44.      TreeNode root = new TreeNode(Integer.valueOf(list[0]));
45.      Stack<TreeNode> stack = new Stack<>();
46.      stack.push(root);
47.
48.      // direction flag
49.      boolean left = true;
50.
51.      for (int i = 1; i < list.length; i++) {
52.        TreeNode node = list[i].equals("#") ? null : new
    TreeNode(Integer.valueOf(list[i]));
53.
54.        if (left) {
55.          stack.peek().left = node;
56.          if (node == null) left = false;
57.        } else {
58.          stack.pop().right = node;
59.          if (node != null) left = true;
60.        }
61.
62.        if (node != null) stack.push(node);
63.      }
64.
```

```
65.        return root;
66.    }
67.
68. }
69.
70. // Your Codec object will be instantiated and called as such:
71. // Codec codec = new Codec();
72. // codec.deserialize(codec.serialize(root));
```

Sum Root to Leaf Numbers

Sum Root to Leaf Numbers

Given a binary tree containing digits from `0-9` only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path `1->2->3` which represents the number `123``.

Find the total sum of all root-to-leaf numbers.

For example,

```
1.      1
2.     / \
3.    2   3
```

The root-to-leaf path `1->2` represents the number `12`.

The root-to-leaf path `1->3` represents the number `13`.

Return the sum = 12 + 13 = `25`.

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
```

```
 5.   *      TreeNode left;
 6.   *      TreeNode right;
 7.   *      TreeNode(int x) { val = x; }
 8.   * }
 9.   */
10.  public class Solution {
11.    int total;
12.
13.    public int sumNumbers(TreeNode root) {
14.      total = 0;
15.      helper(root, 0);
16.      return total;
17.    }
18.
19.    void helper(TreeNode root, int sum) {
20.      if (root == null) return;
21.
22.      sum = sum * 10 + root.val;
23.
24.      if (root.left == null && root.right == null) {
25.        total += sum;
26.        return;
27.      }
28.
29.      helper(root.left, sum);
30.      helper(root.right, sum);
31.    }
32.  }
```

Symmetric Tree

- Symmetric Tree

Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree `[1,2,2,3,4,4,3]` is symmetric:

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

But the following `[1,2,2,null,3,null,3]` is not:

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

Note:

Bonus points if you could solve it both recursively and iteratively.

Solution:

```
1. /**
```

```
 2.    * Definition for a binary tree node.
 3.    * public class TreeNode {
 4.    *      int val;
 5.    *      TreeNode left;
 6.    *      TreeNode right;
 7.    *      TreeNode(int x) { val = x; }
 8.    * }
 9.    */
10.  public class Solution {
11.    public boolean isSymmetric(TreeNode root) {
12.      if (root == null) {
13.        return true;
14.      }
15.      return isSymmetric(root.left, root.right);
16.    }
17.
18.    boolean isSymmetric(TreeNode left, TreeNode right) {
19.      if (left == null && right == null) return true;
20.      if (left == null || right == null) return false;
21.      if (left.val != right.val) return false;
22.      return isSymmetric(left.left, right.right) &&
      isSymmetric(left.right, right.left);
23.    }
24.  }
```

Unique Binary Search Trees

Unique Binary Search Trees

Given n, how many structurally unique **BST**'s (binary search trees) that store values 1…n?

For example,

Given n = 3, there are a total of 5 unique BST's.

```
   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```
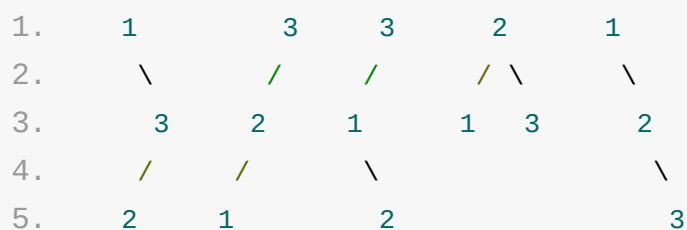
Solution:

```java
public class Solution {
  public int numTrees(int n) {
    if (n == 0)
      return 0;

    int[] dp = new int[n + 1];
    dp[0] = 1;

    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= i; j++) {
        dp[i] += dp[j - 1] * dp[i - j];
      }
    }

```

```
15.        return dp[n];
16.    }
17. }
```

Unique Binary Search Trees II

- Unique Binary Search Trees II

Unique Binary Search Trees II

Given an integer n, generate all structurally unique **BST**'s (binary search trees) that store values 1…n.

For example,

Given n = 3, your program should return all 5 unique BST's shown below.

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

Solution:

```
1.  /**
2.   * Definition for a binary tree node.
3.   * public class TreeNode {
4.   *     int val;
5.   *     TreeNode left;
6.   *     TreeNode right;
7.   *     TreeNode(int x) { val = x; }
8.   * }
9.   */
10. public class Solution {
11.   public List<TreeNode> generateTrees(int n) {
12.       return helper(1, n);
13.   }
```

```
14.
15.    List<TreeNode> helper(int lo, int hi) {
16.      List<TreeNode> res = new ArrayList<>();
17.
18.      if (lo > hi) {
19.        res.add(null);
20.        return res;
21.      }
22.
23.      if (lo == hi) {
24.        res.add(new TreeNode(lo));
25.        return res;
26.      }
27.
28.      for (int i = lo; i <= hi; i++) {
29.        List<TreeNode> left = helper(lo, i - 1);
30.        List<TreeNode> right = helper(i + 1, hi);
31.
32.        for (TreeNode l : left) {
33.          for (TreeNode r : right) {
34.            TreeNode root = new TreeNode(i);
35.            root.left = l;
36.            root.right = r;
37.            res.add(root);
38.          }
39.        }
40.      }
41.
42.      return res;
43.    }
44.  }
```

Validate Binary Search Tree

- Validate Binary Search Tree

Validate Binary Search Tree

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

Assume a BST is defined as follows:

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

Example 1:

```
1.      2
2.     / \
3.    1   3
```

Binary tree `[2,1,3]` , return `true` .

Example 2:

```
1.      1
2.     / \
3.    2   3
```

Binary tree `[1,2,3]` , return `false` .

Solution:

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
  public boolean isValidBST(TreeNode root) {
    return isValidBST(root, null, null);
  }

  boolean isValidBST(TreeNode root, Integer min, Integer max) {
    if (root == null)
      return true;

    if (min != null && root.val <= min)
      return false;

    if (max != null && root.val >= max)
      return false;

    return isValidBST(root.left, min, root.val) &&
    isValidBST(root.right, root.val, max);
  }
}
```

Verify Preorder Sequence in Binary Search Tree

Verify Preorder Sequence in Binary Search Tree

Given an array of numbers, verify whether it is the correct preorder traversal sequence of a binary search tree.

You may assume each number in the sequence is unique.

Follow up:

Could you do it using only constant space complexity?

Solution:

```
1.  public class Solution {
2.    public boolean verifyPreorder(int[] preorder) {
3.      int low = Integer.MIN_VALUE;
4.      Stack<Integer> path = new Stack();
5.
6.      for (int p : preorder) {
7.        if (p < low)
8.          return false;
9.
10.       // If the current value is larger, we are going to the right
11.       // because of preorder, the numbers afterwards cannot be less
12.       // than the ones in the stack
13.       while (!path.empty() && p > path.peek())
14.         low = path.pop();
```

```
15.
16.        path.push(p);
17.      }
18.
19.      return true;
20.    }
21. }
```

Trie

- Trie

Trie

Add and Search Word - Data structure design

Add and Search Word - Data structure design

Design a data structure that supports the following two operations:

```
1.  void addWord(word)
2.  bool search(word)
```

search(word) can search a literal word or a regular expression string containing only letters `a-z` or `.` . A `.` means it can represent any one letter.

For example:

```
1.  addWord("bad")
2.  addWord("dad")
3.  addWord("mad")
4.  search("pad") -> false
5.  search("bad") -> true
6.  search(".ad") -> true
7.  search("b..") -> true
```

Note:

You may assume that all words are consist of lowercase letters `a-z` .

You should be familiar with how a Trie works. If

not, please work on this problem: Implement Trie
(Prefix Tree) first.

Solution:

```
1.   public class WordDictionary {
2.
3.     Trie trie = new Trie();
4.
5.     // Adds a word into the data structure.
6.     public void addWord(String word) {
7.       trie.insert(word);
8.     }
9.
10.    // Returns if the word is in the data structure. A word could
11.    // contain the dot character '.' to represent any one letter.
12.    public boolean search(String word) {
13.      return trie.search(word);
14.    }
15.
16.    class TrieNode {
17.      // Initialize your data structure here.
18.      char c;
19.      boolean isLeaf;
20.      Map<Character, TrieNode> children = new HashMap<Character,
     TrieNode>();
21.
22.      public TrieNode() {}
23.      public TrieNode(char c) { this.c = c; }
24.    }
25.
26.    public class Trie {
27.      private TrieNode root;
28.
29.      public Trie() {
30.        root = new TrieNode();
31.      }
32.
```

```
33.        // Inserts a word into the trie.
34.      public void insert(String word) {
35.        Map<Character, TrieNode> children = root.children;
36.
37.        for (int i = 0; i < word.length(); i++) {
38.          char c = word.charAt(i);
39.
40.          if (!children.containsKey(c))
41.            children.put(c, new TrieNode(c));
42.
43.          TrieNode t = children.get(c);
44.
45.          if (i == word.length() - 1)
46.            t.isLeaf = true;
47.
48.          children = t.children;
49.        }
50.      }
51.
52.      // Returns if the word is in the trie.
53.      public boolean search(String word) {
54.        return dfs(root, word, 0);
55.      }
56.
57.      private boolean dfs(TrieNode node, String word, int index) {
58.        if (node == null || index == word.length() && !node.isLeaf)
59.          return false;
60.
61.        if (node.isLeaf && index == word.length())
62.          return true;
63.
64.        char c = word.charAt(index);
65.
66.        if (c == '.') {
67.          for (char ch = 'a'; ch <= 'z'; ch++) {
68.            if (dfs(node.children.get(ch), word, index + 1))
69.              return true;
70.          }
```

```
71.
72.            return false;
73.        } else {
74.            return dfs(node.children.get(c), word, index + 1);
75.        }
76.    }
77.  }
78. }
79.
80. // Your WordDictionary object will be instantiated and called as
    such:
81. // WordDictionary wordDictionary = new WordDictionary();
82. // wordDictionary.addWord("word");
83. // wordDictionary.search("pattern");
```

Implement Trie

Implement Trie

Implement a trie with `insert` , `search` , and `startsWith` methods.

Note:

You may assume that all inputs are consist of lowercase letters `a-z` .

Solution:

```
class TrieNode {
  char c;
  boolean isLeaf;
  Map<Character, TrieNode> children = new HashMap<>();

  public TrieNode() {}
  public TrieNode(char c) {
    this.c = c;
  }
}

public class Trie {
  private TrieNode root;

  public Trie() {
    root = new TrieNode();
  }

  // Inserts a word into the trie.
  public void insert(String word) {
```

```
21.        Map<Character, TrieNode> children = root.children;
22.
23.        for (int i = 0; i < word.length(); i++) {
24.          char c = word.charAt(i);
25.
26.          if (!children.containsKey(c)) {
27.            children.put(c, new TrieNode(c));
28.          }
29.
30.          if (i == word.length() - 1) {
31.            children.get(c).isLeaf = true;
32.          }
33.
34.          children = children.get(c).children;
35.        }
36.      }
37.
38.      // Returns if the word is in the trie.
39.      public boolean search(String word) {
40.        TrieNode node = getLastTrieNode(word);
41.        return node != null && node.isLeaf;
42.      }
43.
44.      // Returns if there is any word in the trie
45.      // that starts with the given prefix.
46.      public boolean startsWith(String prefix) {
47.        TrieNode node = getLastTrieNode(prefix);
48.        return node != null;
49.      }
50.
51.      private TrieNode getLastTrieNode(String word) {
52.        Map<Character, TrieNode> children = root.children;
53.
54.        for (int i = 0; i < word.length(); i++) {
55.          char c = word.charAt(i);
56.
57.          if (!children.containsKey(c)) {
58.            break;
```

```
59.        }
60.
61.        if (i == word.length() - 1)
62.            return children.get(c);
63.
64.        children = children.get(c).children;
65.      }
66.
67.      return null;
68.    }
69.  }
70.
71.  // Your Trie object will be instantiated and called as such:
72.  // Trie trie = new Trie();
73.  // trie.insert("somestring");
74.  // trie.search("key");
```

Palindrome Pairs

- Palindrome Pairs

Palindrome Pairs

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

Example 1:

Given `words` = `["bat", "tab", "cat"]`

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

The palindromes are `["battab", "tabbat"]`

Example 2:

Given `words` = `["abcd", "dcba", "lls", "s", "sssll"]`

Return `[[0, 1], [1, 0], [3, 2], [2, 4]]`

The palindromes are `["dcbaabcd", "abcddcba", "slls", "llssssll"]`

Word Search II

- Word Search II

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** =

```
1. [
2.   ['o','a','a','n'],
3.   ['e','t','a','e'],
4.   ['i','h','k','r'],
5.   ['i','f','l','v']
6. ]
```

Return `["eat","oath"]` .

Note:

You may assume that all inputs are consist of lowercase letters `a-z` .

Hint:

You would need to optimize your backtracking to pass the larger test. Could you stop backtracking earlier?

If the current candidate does not exist in all words' prefix, you could stop backtracking immediately. What kind of data structure could answer such query efficiently? Does a hash table work? Why or why not? How about a Trie? If you would like to learn how to implement a basic trie, please work on this problem: Implement Trie (Prefix Tree) first.

Solution:

```
1.  public class Solution {
2.    int[] dx = {-1, 1, 0, 0};
3.    int[] dy = {0, 0, -1, 1};
4.
5.    public List<String> findWords(char[][] board, String[] words) {
6.      List<String> res = new ArrayList<String>();
7.
8.      int n = board.length;
9.      int m = board[0].length;
10.
11.     boolean[][] used = new boolean[n][m];
12.
13.     // use a hashset to avoid duplicates
14.     Set<String> set = new HashSet<String>();
15.
16.     // use a trie tree to avoid unnecessary search
17.     Trie trie = new Trie();
18.     for (String word : words)
19.       trie.insert(word);
20.
```

```
21.        // perform dfs from each position
22.        for (int i = 0; i < n; i++)
23.          for (int j = 0; j < m; j++)
24.            search(board, n, m, i, j, "", used, trie, set, res);
25.
26.        return res;
27.      }
28.
29.    void search(char[][] board, int n, int m, int i, int j, String
    word, boolean[][] used, Trie trie, Set<String> set, List<String>
    res) {
30.        if (i < 0 || i >= n || j < 0 || j >= m || used[i][j])
31.          return;
32.
33.        word += board[i][j];
34.
35.        // not the word we are looking for
36.        if (!trie.startsWith(word))
37.          return;
38.
39.        if (!set.contains(word) && trie.search(word)) {
40.          // found the word
41.          set.add(word);
42.          res.add(word);
43.        }
44.
45.        used[i][j] = true;
46.
47.        for (int d = 0; d < 4; d++)
48.          // continue dfs
49.          search(board, n, m, i + dx[d], j + dy[d], word, used, trie,
    set, res);
50.
51.        // backtracking
52.        used[i][j] = false;
53.      }
54.
55.    class TrieNode {
```

```
56.        // Initialize your data structure here.
57.        char c;
58.        boolean isLeaf;
59.        Map<Character, TrieNode> children = new HashMap<Character,
    TrieNode>();
60.
61.        public TrieNode() {}
62.        public TrieNode(char c) { this.c = c; }
63.    }
64.
65.    public class Trie {
66.        private TrieNode root;
67.
68.        public Trie() {
69.            root = new TrieNode();
70.        }
71.
72.        // Inserts a word into the trie.
73.        public void insert(String word) {
74.            Map<Character, TrieNode> children = root.children;
75.
76.            for (int i = 0; i < word.length(); i++) {
77.                char c = word.charAt(i);
78.
79.                if (!children.containsKey(c))
80.                    children.put(c, new TrieNode(c));
81.
82.                TrieNode t = children.get(c);
83.
84.                if (i == word.length() - 1)
85.                    t.isLeaf = true;
86.
87.                children = t.children;
88.            }
89.        }
90.
91.        // Returns if the word is in the trie.
92.        public boolean search(String word) {
```

```
 93.        TrieNode t = searchLastNode(word);
 94.
 95.        return t != null && t.isLeaf;
 96.      }
 97.
 98.      // Returns if there is any word in the trie
 99.      // that starts with the given prefix.
100.      public boolean startsWith(String prefix) {
101.        return searchLastNode(prefix) != null;
102.      }
103.
104.      // Returns the last TrieNode of word
105.      private TrieNode searchLastNode(String word) {
106.        Map<Character, TrieNode> children = root.children;
107.
108.        for (int i = 0; i < word.length(); i++) {
109.          char c = word.charAt(i);
110.
111.          if (!children.containsKey(c))
112.            break;
113.
114.          TrieNode t = children.get(c);
115.
116.          if (i == word.length() - 1)
117.            return t;
118.
119.          children = t.children;
120.        }
121.
122.        return null;
123.      }
124.    }
125.  }
```

Two Pointers

- Two Pointers

Two Pointers

3Sum

3Sum

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: The solution set must not contain duplicate
triplets.

```
1.  For example, given array S = [-1, 0, 1, 2, -1, -4],
2.
3.  A solution set is:
4.  [
5.    [-1, 0, 1],
6.    [-1, -1, 2]
7.  ]
```

Solution:

```java
1.  public class Solution {
2.
3.    public List<List<Integer>> threeSum(int[] num) {
4.      List<List<Integer>> res = new ArrayList<List<Integer>>();
5.
6.      Set<String> set = new HashSet<String>();
7.
8.      Arrays.sort(num);
9.
10.     int n = num.length;
```

```java
11.
12.    for (int i = 0; i < n - 2; i++) {
13.      int j = i + 1, k = n - 1;
14.
15.      while (j < k) {
16.        int sum = num[i] + num[j] + num[k];
17.
18.        if (sum == 0) {
19.          String key = String.format("%d,%d,%d", num[i], num[j],
    num[k]);
20.
21.          if (!set.contains(key)) {
22.            set.add(key);
23.
24.            List<Integer> sol = new ArrayList<Integer>();
25.            sol.add(num[i]);
26.            sol.add(num[j]);
27.            sol.add(num[k]);
28.
29.            res.add(sol);
30.          }
31.
32.          j++;
33.          k--;
34.        } else if (sum < 0) {
35.          j++;
36.        } else {
37.          k--;
38.        }
39.      }
40.    }
41.
42.    return res;
43.  }
44.
45. }
```

Time complexity: O(n^2)

3Sum Closest

- 3Sum Closest

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.

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

Solution:

```java
1.  public class Solution {
2.      public int threeSumClosest(int[] num, int target) {
3.          Arrays.sort(num);
4.
5.          int n = num.length;
6.          int closest = num[0] + num[1] + num[2];
7.
8.          for (int i = 0; i < n - 2; i++) {
9.              int j = i + 1;
10.             int k = n - 1;
11.
12.             while (j < k) {
13.                 int sum = num[i] + num[j] + num[k];
14.
15.                 if (sum == target) {
16.                     return sum;
```

```
17.                   } else if (sum < target) {
18.                       j++;
19.                   } else {
20.                       k--;
21.                   }
22.
23.                   if (Math.abs(target - sum) < Math.abs(target -
    closest)) {
24.                       closest = sum;
25.                   }
26.              }
27.          }
28.
29.          return closest;
30.      }
31. }
```

Time complexity: O(n^2)

3Sum Smaller

- 3Sum Smaller

3Sum Smaller

Given an array of *n* integers *nums* and a *target*, find the number of index triplets `i, j, k` with `0 <= i < j < k < n` that satisfy the condition `nums[i] + nums[j] + nums[k] < target`.

For example, given *nums* = `[-2, 0, 1, 3]`, and *target* = 2.

Return 2. Because there are two triplets which sums are less than 2:

```
1.  [-2, 0, 1]
2.  [-2, 0, 3]
```

Follow up:

Could you solve it in O(n^2) runtime?

Solution:

```
1.  public class Solution {
2.    public int threeSumSmaller(int[] nums, int target) {
3.      Arrays.sort(nums);
4.
5.      int n = nums.length, count = 0;
6.
7.      for (int i = 0; i < n - 2; i++) {
8.        int j = i + 1, k = n - 1;
```

```
 9.
10.        while (j < k) {
11.          int sum = nums[i] + nums[j] + nums[k];
12.
13.          if (sum >= target) {
14.            k--;
15.          } else {
16.            count += k - j;
17.            j++;
18.          }
19.        }
20.      }
21.
22.      return count;
23.    }
24.  }
```

Time complexity: O(n^2)

4Sum

4Sum

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

Note: The solution set must not contain duplicate quadruplets.

```
1.  For example, given array S = [1, 0, -1, 0, -2, 2], and target = 0.
2.
3.  A solution set is:
4.  [
5.    [-1,  0, 0, 1],
6.    [-2, -1, 1, 2],
7.    [-2,  0, 0, 2]
8.  ]
```

Solution:

```
1.  public class Solution {
2.    public List<List<Integer>> fourSum(int[] num, int target) {
3.      List<List<Integer>> res = new ArrayList<List<Integer>>();
4.      Set<String> set = new HashSet<String>();
5.
6.      if (num == null || num.length < 4) {
7.        return res;
8.      }
9.
```

```
10.      Arrays.sort(num);
11.
12.      int n = num.length;
13.
14.      for (int i = 0; i < n - 3; i++) {
15.        for (int j = i + 1; j < n - 2; j++) {
16.          int m = j + 1;
17.          int k = n - 1;
18.
19.          while (m < k) {
20.            int sum = num[i] + num[j] + num[m] + num[k];
21.
22.            if (sum == target) {
23.              String key = String.format("%d,%d,%d,%d", num[i],
    num[j], num[m], num[k]);
24.
25.              if (!set.contains(key)) {
26.                set.add(key);
27.
28.                List<Integer> sol = new ArrayList<Integer>();
29.                sol.add(num[i]);
30.                sol.add(num[j]);
31.                sol.add(num[m]);
32.                sol.add(num[k]);
33.                res.add(sol);
34.              }
35.
36.              m++;
37.              k--;
38.            } else if (sum < target) {
39.              m++;
40.            } else {
41.              k--;
42.            }
43.          }
44.        }
45.      }
46.
```

```
47.      return res;
48.    }
49.  }
```

Time complexity: O(n^3)

Container With Most Water

- Container With Most Water

Container With Most Water

Given *n* non-negative integers *a1, a2, …, an*, where each represents a point at coordinate *(i, ai)*. *n* vertical lines are drawn such that the two endpoints of line i is at *(i, ai)* and *(i, 0)*. Find two lines, which together with x-axis forms a container, such that the container contains the most water.

Note:

You may not slant the container.

Solution:

```
1.  public class Solution {
2.    public int maxArea(int[] height) {
3.      int n = height.length, max = 0;
4.
5.      int i = 0, j = n - 1;
6.
7.      while (i < j) {
8.        int area = Math.min(height[i], height[j]) * (j - i);
9.        max = Math.max(max, area);
10.
11.       if (height[i] < height[j]) {
12.         i++;
13.       } else {
14.         j--;
15.       }
```

```
16.        }
17.
18.        return max;
19.     }
20.  }
```

Find the Duplicate Number

- Find the Duplicate Number

Find the Duplicate Number

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

Note:

- You **must not** modify the array (assume the array is read only).
- You must use only constant, O(1) extra space.
- Your runtime complexity should be less than O(n2).
- There is only one duplicate number in the array, but it could be repeated more than once.

Solution:

```
public class Solution {
  public int findDuplicate(int[] nums) {
    int a = 0, b = 0;

    do {
      a = nums[nums[a]];
      b = nums[b];
    } while (a != b);

```

```
10.        b = 0;
11.
12.     while (a != b) {
13.         a = nums[a];
14.         b = nums[b];
15.     }
16.
17.     return a;
18.   }
19. }
```

Implement strStr

- Implement strStr

Implement strStr

Implement `strStr()` .

Returns the index of the first occurrence of needle in haystack, or `-1` if needle is not part of haystack.

Solution:

```
public class Solution {
  public int strStr(String haystack, String needle) {
    for (int i = 0; ; i++) {
      for (int j = 0; ; j++) {
        if (j == needle.length()) return i;
        if (i + j == haystack.length()) return -1;
        if (needle.charAt(j) != haystack.charAt(i + j)) break;
      }
    }
  }
}
```

Intersection of Two Arrays

- Intersection of Two Arrays

Intersection of Two Arrays

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

Example:

Given nums1 = `[1, 2, 2, 1]` , nums2 = `[2, 2]` , return `[2]` .

Note:

- Each element in the result must be unique.
- The result can be in any order.

Intersection of Two Arrays II

- Intersection of Two Arrays II

Intersection of Two Arrays II

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

Example:

Given nums1 = `[1, 2, 2, 1]` , nums2 = `[2, 2]` , return `[2, 2]` .

Note:

- Each element in the result should appear as many times as it shows in both arrays.
- The result can be in any order.

Follow up:

- What if the given array is already sorted? How would you optimize your algorithm?
- What if nums1's size is small compared to nums2's size? Which algorithm is better?
- What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?

Linked List Cycle

- Linked List Cycle

Linked List Cycle

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

Follow up:

Can you solve it without using extra space?

Solution:

```
 1.  /**
 2.   * Definition for singly-linked list.
 3.   * class ListNode {
 4.   *     int val;
 5.   *     ListNode next;
 6.   *     ListNode(int x) {
 7.   *         val = x;
 8.   *         next = null;
 9.   *     }
10.   * }
11.   */
12.  public class Solution {
13.    public boolean hasCycle(ListNode head) {
14.      ListNode slow = head, fast = head;
15.
16.      while (fast != null && fast.next != null) {
17.        slow = slow.next;
18.        fast = fast.next.next;
19.
20.        if (slow == fast)
21.          return true;
```

```
22.        }
23.
24.       return false;
25.    }
26. }
```

Linked List Cycle II

- Linked List Cycle II

Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return `null` .

Note: Do not modify the linked list.

Follow up:

Can you solve it without using extra space?

Solution:

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
  public ListNode detectCycle(ListNode head) {
    ListNode slow = head, fast = head;

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

```

```
20.          if (slow == fast)
21.            break;
22.        }
23.
24.      if (fast == null || fast.next == null)
25.        return null;
26.
27.      slow = head;
28.      while (slow != fast) {
29.        slow = slow.next;
30.        fast = fast.next;
31.      }
32.
33.      return slow;
34.    }
35.  }
```

Longest Substring with At Most Two Distinct Characters

-

Longest Substring with At Most Two Distinct Characters

Given a string, find the length of the longest substring T that contains at most 2 distinct characters.

For example, Given s = `"eceba"` ,

T is `"ece"` which its length is 3.

Solution:

```
1.  public class Solution {
2.    public int lengthOfLongestSubstringTwoDistinct(String s) {
3.      if (s == null) return 0;
4.
5.      int i = 0; // slow pointer
6.      int j = 0; // fast pointer
7.      int max = 0;
8.
9.      // use a map to track the char and its count
10.     Map<Character, Integer> map = new HashMap<Character, Integer>();
11.
12.     while (j < s.length()) {
13.       char ch = s.charAt(j);
14.
15.       if (map.size() < 2 || map.containsKey(ch)) {
```

```
16.         // less than 2 distinct chars or the char is in the map
    already
17.         // put it to the map and update the count
18.         map.put(ch, map.containsKey(ch) ? map.get(ch) + 1: 1);
19.
20.         // update the max
21.         max = Math.max(max, j - i + 1);
22.
23.         j++;
24.       } else {
25.         // we keep removing the old chars from the map
26.         // till we only have one distinct char
27.         while (map.size() == 2) {
28.           ch = s.charAt(i);
29.
30.           map.put(ch, map.get(ch) - 1);
31.
32.           if (map.get(ch) == 0)
33.             map.remove(ch);
34.
35.           i++;
36.         }
37.       }
38.     }
39.
40.     return max;
41.   }
42. }
```

Longest Substring Without Repeating Characters

- Longest Substring Without Repeating Characters

Longest Substring Without Repeating Characters

Given a string, find the length of the **longest substring** without repeating characters.

Examples:

Given `"abcabcbb"`, the answer is `"abc"`, which the length is `3`.

Given `"bbbbb"`, the answer is `"b"`, with the length of `1`.

Given `"pwwkew"`, the answer is `"wke"`, with the length of `3`. Note that the answer must be a **substring**, `"pwke"` is a subsequence and not a substring.

Solution:

```
1.  public class Solution {
2.    public int lengthOfLongestSubstring(String s) {
3.      int i = 0, j = 0, max = 0;
4.      Set<Character> set = new HashSet<>();
5.
6.      while (j < s.length()) {
7.        if (!set.contains(s.charAt(j))) {
8.          set.add(s.charAt(j++));
9.          max = Math.max(max, set.size());
```

```
10.        } else {
11.           set.remove(s.charAt(i++));
12.        }
13.     }
14.
15.     return max;
16.   }
17. }
```

Merge Sorted Array

- Merge Sorted Array

Merge Sorted Array

Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

Note:

You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2. The number of elements initialized in nums1 and nums2 are m and n respectively.

Solution:

```java
public class Solution {
  public void merge(int[] nums1, int m, int[] nums2, int n) {
    int[] copy = new int[m];

    for (int k = 0; k < m; k++)
      copy[k] = nums1[k];

    int i = 0, j = 0;

    for (int k = 0; k < m + n; k++) {
      if (i >= m)                     nums1[k] = nums2[j++];
      else if (j >= n)                nums1[k] = copy[i++];
      else if (copy[i] < nums2[j])    nums1[k] = copy[i++];
      else                            nums1[k] = nums2[j++];
    }
```

```
16.    }
17. }
```

Minimum Size Subarray Sum

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 ≥ 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 under the problem constraint.

More practice:

If you have figured out the O(n) solution, try coding another solution of which the time complexity is O(n log n).

Solution:

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

    // i is slow pointer, j is fast pointer
    int i = 0, j = 0, sum = 0, min = Integer.MAX_VALUE;

    while (j < a.length) {
```

```
10.         sum += a[j++];
11.
12.         while (sum >= s) {
13.           min = Math.min(min, j - i);
14.           sum -= a[i++];
15.         }
16.       }
17.
18.       return min == Integer.MAX_VALUE ? 0 : min;
19.   }
20. }
```

Minimum Window Substring

- Minimum Window Substring

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"` .

Note:

If there is no such window in S that covers all characters in T, return the empty string `""` .

If there are multiple such windows, you are guaranteed that there will always be only one unique minimum window in S.

Solution:

```
1.  public class Solution {
2.    public String minWindow(String s, String t) {
3.      // step 1. create a hashmap for t
4.      char[] chsT = t.toCharArray();
5.      char[] chsS = s.toCharArray();
```

```
  6.
  7.      Map<Character, Integer> map = new HashMap<Character, Integer>
   ();
  8.
  9.      for (int i = 0; i < chsT.length; i++) {
 10.        char ch = chsT[i];
 11.        int count = map.containsKey(ch) ? map.get(ch) : 0;
 12.        map.put(ch, count + 1);
 13.      }
 14.
 15.      // step 2. use two pointers
 16.      int i = 0, j = 0, count = 0;
 17.      String res = "";
 18.
 19.      while (j < s.length()) {
 20.        char ch_j = chsS[j];
 21.
 22.        if (map.containsKey(ch_j)) {
 23.          // j find a character, update the count and the map
 24.          if (map.get(ch_j) > 0) {
 25.              count++;
 26.          }
 27.          map.put(ch_j, map.get(ch_j) - 1);
 28.
 29.          while (count == t.length()) {
 30.            // count the length of current substring
 31.            if (res.equals("") || j - i + 1 < res.length()) {
 32.              res = s.substring(i, j + 1);
 33.            }
 34.
 35.            char ch_i = chsS[i];
 36.
 37.            if (map.containsKey(ch_i)) {
 38.              // i find a character, update the count
 39.              if (map.get(ch_i) >= 0) {
 40.                count--;
 41.              }
 42.              map.put(ch_i, map.get(ch_i) + 1);
```

```
43.              }
44.
45.              i++;
46.          }
47.        }
48.
49.        j++;
50.      }
51.
52.      return res;
53.    }
54.  }
```

Move Zeroes

- Move Zeroes

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]`.

Note:

1. You must do this in-place without making a copy of the array.
2. Minimize the total number of operations.

Solution:

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

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

```

```
13.    void swap(int[] nums, int i, int j) {
14.      int tmp = nums[i];
15.      nums[i] = nums[j];
16.      nums[j] = tmp;
17.    }
18.  }
```

Palindrome Linked List

- Palindrome Linked List

Palindrome Linked List

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

Follow up:

Could you do it in O(n) time and O(1) space?

Solution:

```
1.  /**
2.   * Definition for singly-linked list.
3.   * public class ListNode {
4.   *     int val;
5.   *     ListNode next;
6.   *     ListNode(int x) { val = x; }
7.   * }
8.   */
9.  public class Solution {
10.   public boolean isPalindrome(ListNode head) {
11.     if (head == null || head.next == null)
12.       return true;
13.
14.     // step 1. cut the original list to two halves
15.     ListNode prev = null, slow = head, fast = head, l1 = head;
16.
17.     while (fast != null && fast.next != null) {
18.       prev = slow;
19.       slow = slow.next;
20.       fast = fast.next.next;
21.     }
```

```
22.
23.        prev.next = null;
24.
25.        // step 2. reverse the 2nd half
26.        ListNode l2 = (fast == null) ? reverse(slow) :
      reverse(slow.next);
27.
28.        // step 3. compare the new two halves
29.        while (l1 != null && l2 != null) {
30.          if (l1.val != l2.val)
31.            return false;
32.
33.          l1 = l1.next;
34.          l2 = l2.next;
35.        }
36.
37.        return true;
38.      }
39.
40.      // helper function: reverse a list
41.      ListNode reverse(ListNode head) {
42.        ListNode prev = null, curr = head, next = null;
43.
44.        while (curr != null) {
45.          next = curr.next;
46.          curr.next = prev;
47.          prev = curr;
48.          curr = next;
49.        }
50.
51.        return prev;
52.      }
53.  }
```

Partition List

- Partition List

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` .

Solution:

```
1.  /**
2.   * Definition for singly-linked list.
3.   * public class ListNode {
4.   *     int val;
5.   *     ListNode next;
6.   *     ListNode(int x) { val = x; }
7.   * }
8.   */
9.  public class Solution {
10.    public ListNode partition(ListNode head, int x) {
11.      if (head == null) {
12.        return null;
13.      }
14.
```

```
15.       // create fake heads
16.       ListNode f1 = new ListNode(0);
17.       ListNode f2 = new ListNode(0);
18.
19.       ListNode p1 = f1;
20.       ListNode p2 = f2;
21.
22.       ListNode node = head;
23.       while (node != null) {
24.         if (node.val < x) {
25.           p1.next = node;
26.           p1 = p1.next;
27.         } else {
28.           p2.next = node;
29.           p2 = p2.next;
30.         }
31.         node = node.next;
32.       }
33.
34.       p1.next = f2.next;
35.       p2.next = null;
36.
37.       return f1.next;
38.     }
39.   }
```

Remove Duplicates from Sorted Array

- Remove Duplicates from Sorted Array

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 `nums = [1,1,2]` ,

Your function should return length = `2` , with the first two elements of nums being `1` and `2` respectively. It doesn't matter what you leave beyond the new length.

Solution:

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

    int i = 0;

    for (int j = 1; j < nums.length; j++) {
      if (nums[j] != nums[i]) {
        nums[++i] = nums[j];
```

```
12.            }
13.        }
14.
15.        return i + 1;
16.    }
17. }
```

Remove Duplicates from Sorted Array II

- Remove Duplicates from Sorted Array II

Remove Duplicates from Sorted Array II

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

For example,
Given sorted array nums = `[1,1,1,2,2,3]` ,

Your function should return length = 5, with the first five elements of nums being 1, 1, 2, 2 and 3. It doesn't matter what you leave beyond the new length.

Solution:

```java
public class Solution {
  public int removeDuplicates(int[] nums) {
    if (nums == null) return 0;
    if (nums.length < 3) return nums.length;

    int n = nums.length;

    int i = 2;

    for (int j = 2; j < n; j++) {
      if (nums[j] != nums[i - 2]) {
        nums[i++] = nums[j];
      }
    }
```

```
15.
16.      return i;
17.   }
18. }
```

Remove Element

- Remove Element

Remove Element

Given an array and a value, remove all instances of that value in place and return the new length.

Do not allocate extra space for another array, you must do this in place with constant memory.

The order of elements can be changed. It doesn't matter what you leave beyond the new length.

Example:

Given input array nums = `[3,2,2,3]` , val = `3`

Your function should return length = 2, with the first two elements of nums being 2.

Hint:

1. Try two pointers.
2. Did you use the property of "the order of elements can be changed"?
3. What happens when the elements to remove are rare?

Solution:

```
1.  public class Solution {
```

```java
 2.    public int removeElement(int[] nums, int val) {
 3.      if (nums == null) {
 4.        return 0;
 5.      }
 6.
 7.      int i = 0;
 8.      for (int j = 0; j < nums.length; j++) {
 9.        if (nums[j] != val) {
10.          nums[i++] = nums[j];
11.        }
12.      }
13.
14.      return i;
15.    }
16.  }
```

Remove Nth Node From End of List

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,

```
1.    Given linked list: 1->2->3->4->5, and n = 2.
2.
3.    After removing the second node from the end, the linked list
      becomes 1->2->3->5.
```

Note:

Given n will always be valid.

Try to do this in one pass.

Solution:

```
1.  /**
2.   * Definition for singly-linked list.
3.   * public class ListNode {
4.   *     int val;
5.   *     ListNode next;
6.   *     ListNode(int x) { val = x; }
7.   * }
8.   */
9.  public class Solution {
10.    public ListNode removeNthFromEnd(ListNode head, int n) {
11.      ListNode slow = head, fast = head;
```

```
12.
13.      while (n-- > 0)
14.        fast = fast.next;
15.
16.      if (fast == null)
17.        return head.next;
18.
19.      while (fast != null && fast.next != null) {
20.        slow = slow.next;
21.        fast = fast.next;
22.      }
23.
24.      slow.next = slow.next.next;
25.
26.      return head;
27.    }
28.  }
```

Reverse String

-

Reverse String

Write a function that takes a string as input and returns the string reversed.

Example:

Given s = "hello", return "olleh".

Reverse Vowels of a String

- Reverse Vowels of a String

Reverse Vowels of a String

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

Example 1:

Given s = "hello", return "holle".

Example 2:

Given s = "leetcode", return "leotcede".

Rotate List

- Rotate List

Rotate List

Given a list, rotate the list to the right by k places, where k is non-negative.

For example:

Given `1->2->3->4->5->NULL` and k = `2`,

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

Solution:

```java
1.  /**
2.   * Definition for singly-linked list.
3.   * public class ListNode {
4.   *     int val;
5.   *     ListNode next;
6.   *     ListNode(int x) { val = x; }
7.   * }
8.   */
9.  public class Solution {
10.    public ListNode rotateRight(ListNode head, int k) {
11.      if (head == null) {
12.        return null;
13.      }
14.
15.      int len = getLength(head);
16.      k %= len;
17.
18.      if (k == 0) {
19.        return head;
```

```
20.        }
21.
22.        ListNode fast = head;
23.        ListNode slow = head;
24.
25.        while (fast != null && k > 0) {
26.          fast = fast.next;
27.          k--;
28.        }
29.
30.        while (fast.next != null) {
31.          slow = slow.next;
32.          fast = fast.next;
33.        }
34.
35.        fast.next = head;
36.        head = slow.next;
37.        slow.next = null;
38.
39.        return head;
40.    }
41.
42.    int getLength(ListNode head) {
43.        int len = 0;
44.        while (head != null) {
45.          head = head.next;
46.          len++;
47.        }
48.        return len;
49.    }
50. }
```

Sort Colors

Sort Colors

Given an array with n objects colored red, white or blue, sort them so that objects of the same color are adjacent, with the colors in the order red, white and blue.

Here, we will use the integers 0, 1, and 2 to represent the color red, white, and blue respectively.

Note:

You are not suppose to use the library's sort function for this problem.

Follow up:

A rather straight forward solution is a two-pass algorithm using counting sort.

First, iterate the array counting number of 0's, 1's, and 2's, then overwrite array with total number of 0's, then 1's and followed by 2's.

Could you come up with an one-pass algorithm using only constant space?

Solution:

```java
public class Solution {
  public void sortColors(int[] nums) {
    if (nums == null) {
      return;
    }

    int n = nums.length;
    int i = -1, j = 0, k = n;

    while (j < k) {
      if (nums[j] == 1) {
        j++;
      } else if (nums[j] == 0) {
        swap(nums, ++i, j++);
      } else {
        swap(nums, --k, j);
      }
    }
  }

  void swap(int[] nums, int i, int j) {
    int tmp = nums[i];
    nums[i] = nums[j];
    nums[j] = tmp;
  }
}
```

Sort Transformed Array

- Sort Transformed Array

Sort Transformed Array

Given a sorted array of integers nums and integer values a, b and c. Apply a function of the form $f(x) = ax^2 + bx + c$ to each element x in the array.

The returned array must be in sorted order.

Expected time complexity: O(n)

Example:

```
1.  nums = [-4, -2, 2, 4], a = 1, b = 3, c = 5,
2.
3.  Result: [3, 9, 15, 33]
4.
5.  nums = [-4, -2, 2, 4], a = -1, b = 3, c = 5
6.
7.  Result: [-23, -5, 1, 7]
```

Substring with Concatenation of All Words

- Substring with Concatenation of All Words

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"]`

You should return the indices: `[0,9]` . (order does not matter).

Solution:

```java
public class Solution {
  public List<Integer> findSubstring(String s, String[] words) {
    List<Integer> res = new ArrayList<Integer>();

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

    // build the hash map
    Map<String, Integer> map = new HashMap<String, Integer>();
```

```
11.      for (int i = 0; i < words.length; i++) {
12.        String word = words[i];
13.
14.        int count = map.containsKey(word) ? map.get(word) + 1 : 1;
15.        map.put(word, count);
16.      }
17.
18.      for (int i = 0; i < s.length() - len + 1; i++) {
19.        Map<String, Integer> temp = new HashMap<String, Integer>
     (map);
20.
21.        for (int j = i; j < i + len; j += n) {
22.          String str = s.substring(j, j + n);
23.
24.          if (temp.containsKey(str)) {
25.            int count = temp.get(str) - 1;
26.
27.            if (count == 0) {
28.              temp.remove(str);
29.            } else {
30.              temp.put(str, count);
31.            }
32.          } else {
33.            break;
34.          }
35.        }
36.
37.        if (temp.size() == 0) {
38.          res.add(i);
39.        }
40.      }
41.
42.      return res;
43.    }
44. }
```

Trapping Rain Water

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.

The above elevation map is represented by array `[0,1,0,2,1,0,1,3,2,1,2,1]` . In this case, 6 units of rain water (blue section) are being trapped.

Solution:

```
1.  public class Solution {
2.    public int trap(int[] height) {
3.      if (height == null || height.length == 0) {
4.        return 0;
5.      }
6.
```

```
 7.        int n = height.length;
 8.        int max = 0;
 9.
10.        // scan from left
11.        int[] lmax = new int[n];
12.        for (int i = 0; i < n; i++) {
13.          lmax[i] = max;
14.          max = Math.max(max, height[i]);
15.        }
16.
17.        // scan from right
18.        max = 0;
19.        int[] rmax = new int[n];
20.        for (int i = n - 1; i >= 0; i--) {
21.          rmax[i] = max;
22.          max = Math.max(max, height[i]);
23.        }
24.
25.        // final scan
26.        int res = 0;
27.        for (int i = 0; i < n; i++) {
28.          int water = Math.min(lmax[i], rmax[i]) - height[i];
29.          res += water >= 0 ? water : 0;
30.        }
31.
32.        return res;
33.    }
34. }
```

Two Sum II - Input array is sorted

- Two Sum II - Input array is sorted

Two Sum II - Input array is sorted

Valid Palindrome

- Valid Palindrome

Valid Palindrome

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

For example,

`"A man, a plan, a canal: Panama"` is a palindrome.

`"race a car"` is not a palindrome.

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.

Solution:

```java
public class Solution {
  public boolean isPalindrome(String s) {
    if (s == null) return false;

    s = s.toLowerCase();

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

    char[] chars = s.toCharArray();
```

```
10.
11.      while (i < j) {
12.        // skip any non-alphanumeric chars
13.        while (i < n && !Character.isLetterOrDigit(chars[i])) { i++;
    }
14.        while (j >= 0 && !Character.isLetterOrDigit(chars[j])) { j--;
    }
15.
16.        if (i < j && chars[i++] != chars[j--]) {
17.          return false;
18.        }
19.      }
20.
21.      return true;
22.    }
23. }
```

Union Find

- Union Find

Union Find

Graph Valid Tree

- Graph Valid Tree

Graph Valid Tree

Given `n` nodes labeled from `0` to `n - 1` and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

For example:

Given `n = 5` and edges = `[[0, 1], [0, 2], [0, 3], [1, 4]]`, return `true`.

Given `n = 5` and edges = `[[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]]`, return `false`.

Hint:

1. Given `n = 5` and edges = `[[0, 1], [1, 2], [3, 4]]`, what should your return? Is this case a valid tree?
2. According to the definition of tree on Wikipedia: "a tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree."

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

together in edges.

Solution:

```java
public class Solution {
  public boolean validTree(int n, int[][] edges) {
    int[] nums = new int[n];

    Arrays.fill(nums, -1);

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

      x = find(nums, x);
      y = find(nums, y);

      if (x == y)
        return false;

      nums[x] = y;
    }

    return edges.length == n - 1;
  }

  int find(int[] nums, int i) {
    if (nums[i] == -1)
      return i;

    return find(nums, nums[i]);
  }
}
```

Longest Consecutive Sequence

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 is `[1, 2, 3, 4]`. Return its length: `4`.

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

Solution:

```
1.  public class Solution {
2.    public int longestConsecutive(int[] nums) {
3.      int max = 0;
4.
5.      Set<Integer> set = new HashSet<Integer>();
6.
7.      for (int i = 0; i < nums.length; i++) {
8.        set.add(nums[i]);
9.      }
10.
11.     for (int i = 0; i < nums.length; i++) {
12.       int count = 1;
13.
14.       // look left
15.       int num = nums[i];
16.       while (set.contains(--num)) {
17.         count++;
```

```
18.          set.remove(num);
19.        }
20.
21.        // look right
22.        num = nums[i];
23.        while (set.contains(++num)) {
24.          count++;
25.          set.remove(num);
26.        }
27.
28.        max = Math.max(max, count);
29.      }
30.
31.      return max;
32.    }
33. }
```

Number of Connected Components in an Undirected Graph

- Number of Connected Components in an Undirected Graph

Number of Connected Components in an Undirected Graph

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

Example 1:

```
1.      0        3
2.      |        |
3.      1 --- 2   4
```

Given `n = 5` and `edges = [[0, 1], [1, 2], [3, 4]]`, return `2`.

Example 2:

```
1.      0          4
2.      |          |
3.      1 --- 2 --- 3
```

Given `n = 5` and `edges = [[0, 1], [1, 2], [2, 3], [3, 4]]`, return `1`.

Note:

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

Number of Islands

- Number of Islands

Number of Islands

Given a 2d 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:

```
1.  11110
2.  11010
3.  11000
4.  00000
```

Answer: 1

Example 2:

```
1.  11000
2.  11000
3.  00100
4.  00011
```

Answer: 3

Number of Islands II

- Number of Islands II

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.

Example:

Given `m = 3, n = 3` , `positions = [[0,0], [0,1], [1,2], [2,1]]` .

Initially, the 2d grid `grid` is filled with water. (Assume 0 represents water and 1 represents land).

```
1. 0 0 0
2. 0 0 0
3. 0 0 0
```

Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land.

```
1. 1 0 0
2. 0 0 0    Number of islands = 1
```

```
3.  0 0 0
```

Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land.

```
1.  1 1 0
2.  0 0 0    Number of islands = 1
3.  0 0 0
```

Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land.

```
1.  1 1 0
2.  0 0 1    Number of islands = 2
3.  0 0 0
```

Operation #4: addLand(2, 1) turns the water at grid[2][1] into a land.

```
1.  1 1 0
2.  0 0 1    Number of islands = 3
3.  0 1 0
```

We return the result as an array: `[1, 1, 2, 3]`

Challenge:

Can you do it in time complexity O(k log mn), where k is the length of the `positions` ?

Solution:

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

```
 4.
 5.    public List<Integer> numIslands2(int m, int n, int[][] positions)
   {
 6.        List<Integer> res = new ArrayList<>();
 7.
 8.        int count = 0;
 9.        int[] nums = new int[m*n];
10.        Arrays.fill(nums, -1);
11.
12.        for (int[] p : positions) {
13.          // for each position, mark it as new island
14.          int x = p[0]*n + p[1];
15.          nums[x] = x;
16.          count++;
17.
18.          for (int i = 0; i < 4; i++) {
19.            // check neighbours
20.            int nx = p[0] + dx[i];
21.            int ny = p[1] + dy[i];
22.            int y = nx*n + ny;
23.
24.            // ignore invalid position
25.            if (nx < 0 || nx >= m || ny < 0 || ny >= n || nums[y] ==
   -1) {
26.                continue;
27.            }
28.
29.            // find and union islands
30.            y = find(nums, y);
31.            x = find(nums, x);
32.            nums[y] = x;
33.
34.            // merge two isolated islands
35.            if (y != x) {
36.                count--;
37.            }
38.          }
39.
```

```
40.        res.add(count);
41.      }
42.
43.      return res;
44.    }
45.
46.    int find(int nums[], int i) {
47.      if (nums[i] == i) {
48.        return i;
49.      }
50.
51.      nums[i] = find(nums, nums[i]);
52.      return nums[i];
53.    }
54.  }
```

Surrounded Regions

- Surrounded Regions

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,

```
1.  X X X X
2.  X O O X
3.  X X O X
4.  X O X X
```

After running your function, the board should be:

```
1.  X X X X
2.  X X X X
3.  X X X X
4.  X O X X
```

Solution:

```
1.  public class Solution {
2.    int[] dx = {-1, 1, 0, 0};
3.    int[] dy = {0, 0, -1, 1};
4.
5.    public void solve(char[][] board) {
6.      if (board == null || board.length == 0 || board[0].length == 0)
    return;
```

```
 7.
 8.      int n = board.length;
 9.      int m = board[0].length;
10.
11.      // scan the borders and mark the 'O's to '1'
12.      for (int i = 0; i < n; i++)
13.        for (int j = 0; j < m; j++)
14.          if ((i == 0 || i == n - 1 || j == 0 || j == m - 1) &&
    board[i][j] == 'O')
15.            bfs(board, n, m, i, j);
16.
17.      // scan the inner area and mark the 'O's to 'X'
18.      for (int i = 1; i < n; i++)
19.        for (int j = 1; j < m; j++)
20.          if (board[i][j] == 'O')
21.            board[i][j] = 'X';
22.
23.      // reset all the '1's to 'O's
24.      for (int i = 0; i < n; i++)
25.        for (int j = 0; j < m; j++)
26.          if (board[i][j] == '1')
27.            board[i][j] = 'O';
28.
29.    }
30.
31.    void bfs(char[][] board, int n, int m, int i, int j) {
32.      Queue<int[]> queue = new LinkedList<int[]>();
33.      queue.add(new int[]{i, j});
34.
35.      board[i][j] = '1';
36.
37.      while (!queue.isEmpty()) {
38.        int[] pos = queue.poll();
39.
40.        for (int k = 0; k < 4; k++) {
41.          i = pos[0] + dx[k];
42.          j = pos[1] + dy[k];
43.
```

```
44.          if (i >= 0 && i < n && j >= 0 && j < m && board[i][j] ==
     'O') {
45.              board[i][j] = '1';
46.              queue.add(new int[]{i, j});
47.          }
48.        }
49.      }
50.    }
51. }
```