Contents

top-k-frequent-elements.py

```
# Example 2:
#
# Input: nums = [1], k = 1
# Output: [1]# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
```

```python
    def topKFrequent(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: List[int]
        """
        counts = collections.Counter(nums)
        buckets = [[] for _ in xrange(len(nums)+1)]
        for i, count in counts.iteritems():
            buckets[count].append(i)

        result = []
        for i in reversed(xrange(len(buckets))):
            for j in xrange(len(buckets[i])):
                result.append(buckets[i][j])
                if len(result) == k:
                    return result
        return result


# Time:  O(n) ~ O(n^2), O(n) on average.
# Space: O(n)
# Quick Select Solution
from random import randint
class Solution2(object):
    def topKFrequent(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: List[int]
        """
        counts = collections.Counter(nums)
        p = []
        for key, val in counts.iteritems():
            p.append((-val, key))
        self.kthElement(p, k-1)

        result = []
        for i in xrange(k):
            result.append(p[i][1])
        return result

    def kthElement(self, nums, k):
        def PartitionAroundPivot(left, right, pivot_idx, nums):
            pivot_value = nums[pivot_idx]
            new_pivot_idx = left
            nums[pivot_idx], nums[right] = nums[right], nums[pivot_idx]
            for i in xrange(left, right):
                if nums[i] < pivot_value:
                    nums[i], nums[new_pivot_idx] = nums[new_pivot_idx], nums[i]
                    new_pivot_idx += 1

            nums[right], nums[new_pivot_idx] = nums[new_pivot_idx], nums[right]
            return new_pivot_idx

        left, right = 0, len(nums) - 1
        while left <= right:
            pivot_idx = randint(left, right)
            new_pivot_idx = PartitionAroundPivot(left, right, pivot_idx, nums)
            if new_pivot_idx == k:
```

```python
                return
            elif new_pivot_idx > k:
                right = new_pivot_idx - 1
            else:  # new_pivot_idx < k.
                left = new_pivot_idx + 1


# Time:  O(nlogk)
# Space: O(n)
class Solution3(object):
    def topKFrequent(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: List[int]
        """
        return [key for key, _ in collections.Counter(nums).most_common(k)]
```

reconstruct-original-digits-from-english.py

```python
# Given a non-empty string containing an out-of-order English representation of
# digits 0-9, output the digits in ascending order.
#
# Note:
#
#
# Input contains only lowercase English letters.
# Input is guaranteed to be valid and can be transformed to its original digits.
# That means invalid inputs such as "abc" or "zerone" are not permitted.
# Input length is less than 50,000.
#
#
#
# Example 1:
#
# Input: "owoztneoer"
#
# Output: "012"
#
#
#
# Example 2:
#
# Input: "fviefuro"
#
# Output: "45"# Time:  O(n)
# Space: O(1)

from collections import Counter

class Solution(object):
    def originalDigits(self, s):
        """
        :type s: str
        :rtype: str
        """
        # The count of each char in each number string.
        cnts = [Counter(_) for _ in ["zero", "one", "two", "three", \
                                     "four", "five", "six", "seven", \
                                     "eight", "nine"]]

        # The order for greedy method.
        order = [0, 2, 4, 6, 8, 1, 3, 5, 7, 9]

        # The unique char in the order.
        unique_chars = ['z', 'o', 'w', 't', 'u', \
                        'f', 'x', 's', 'g', 'n']

        cnt = Counter(list(s))
        res = []
        for i in order:
            while cnt[unique_chars[i]] > 0:
                cnt -= cnts[i]
                res.append(i)
        res.sort()

        return "".join(map(str, res))
```

compare-version-numbers.py

```python
# 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.
# You may assume the default revision number for each level of a version number
# to be 0. For example, version number 3.4 has a revision number of 3 and 4 for
# its first and second level revision number. Its third and fourth level revision
# number are both 0.
#
#
#
# Example 1:
# Input: version1 = "0.1", version2 = "1.1"
# Output: -1
#
# Example 2:
# Input: version1 = "1.0.1", version2 = "1"
# Output: 1
#
# Example 3:
# Input: version1 = "7.5.2.4", version2 = "7.5.3"
# Output: -1
#
# Example 4:
# Input: version1 = "1.01", version2 = "1.001"
# Output: 0
# Explanation: Ignoring leading zeroes, both "01" and "001" represent the same
# number "1"
#
# Example 5:
# Input: version1 = "1.0", version2 = "1.0.0"
# Output: 0
# Explanation: The first version number does not have a third level revision
# number, which means its third level revision number is default to "0"
#
#
#
# Note:
#
# Version strings are composed of numeric strings separated by dots . and this
# numeric strings may have leading zeroes.
# Version strings do not start or end with dots, and they will not be two
# consecutive dots.# Time:  O(n)
# Space: O(1)

import itertools


class Solution(object):
    def compareVersion(self, version1, version2):
        """
```

```python
        :type version1: str
        :type version2: str
        :rtype: int
        """
        n1, n2 = len(version1), len(version2)
        i, j = 0, 0
        while i < n1 or j < n2:
            v1, v2 = 0, 0
            while i < n1 and version1[i] != '.':
                v1 = v1 * 10 + int(version1[i])
                i += 1
            while j < n2 and version2[j] != '.':
                v2 = v2 * 10 + int(version2[j])
                j += 1
            if v1 != v2:
                return 1 if v1 > v2 else -1
            i += 1
            j += 1

        return 0

# Time:  O(n)
# Space: O(n)


class Solution2(object):
    def compareVersion(self, version1, version2):
        """
        :type version1: str
        :type version2: str
        :rtype: int
        """
        v1, v2 = version1.split("."), version2.split(".")

        if len(v1) > len(v2):
            v2 += ['0' for _ in xrange(len(v1) - len(v2))]
        elif len(v1) < len(v2):
            v1 += ['0' for _ in xrange(len(v2) - len(v1))]

        i = 0
        while i < len(v1):
            if int(v1[i]) > int(v2[i]):
                return 1
            elif int(v1[i]) < int(v2[i]):
                return -1
            else:
                i += 1

        return 0

    def compareVersion2(self, version1, version2):
        """
        :type version1: str
        :type version2: str
        :rtype: int
        """
        v1 = [int(x) for x in version1.split('.')]
        v2 = [int(x) for x in version2.split('.')]
        while len(v1) != len(v2):
            if len(v1) > len(v2):
```

```
            v2.append(0)
        else:
            v1.append(0)
    return cmp(v1, v2)

def compareVersion3(self, version1, version2):
    splits = (map(int, v.split('.')) for v in (version1, version2))
    return cmp(*zip(*itertools.izip_longest(*splits, fillvalue=0)))

def compareVersion4(self, version1, version2):
    main1, _, rest1 = ('0' + version1).partition('.')
    main2, _, rest2 = ('0' + version2).partition('.')
    return cmp(int(main1), int(main2)) or len(rest1 + rest2) and self.compareVersion4(rest1, rest2)
```

combination-sum-iv.py

```python
# Given an integer array with all positive numbers and no duplicates, find the
# number of possible combinations that add up to a positive integer target.
#
# Example:
#
# nums = [1, 2, 3]
# target = 4
#
# The possible combination ways are:
# (1, 1, 1, 1)
# (1, 1, 2)
# (1, 2, 1)
# (1, 3)
# (2, 1, 1)
# (2, 2)
# (3, 1)
#
# Note that different sequences are counted as different combinations.
#
# Therefore the output is 7.
#
#
#
#
# Follow up:
#
# What if negative numbers are allowed in the given array?
#
# How does it change the problem?
#
# What limitation we need to add to the question to allow negative numbers?
#
# Credits:
#
# Special thanks to @pbrother for adding this problem and creating all test
# cases.# Time:  O(nlon + n * t), t is the value of target.
# Space: O(t)

class Solution(object):
    def combinationSum4(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        dp = [0] * (target+1)
        dp[0] = 1
        nums.sort()

        for i in xrange(1, target+1):
            for j in xrange(len(nums)):
                if nums[j] <= i:
                    dp[i] += dp[i - nums[j]]
                else:
                    break

        return dp[target]
```

koko-eating-bananas.py

```python
# Koko loves to eat bananas.  There are N piles of bananas, the i-th pile has
# piles[i] bananas.  The guards have gone and will come back in H hours.
#
# Koko can decide her bananas-per-hour eating speed of K.  Each hour, she
# chooses some pile of bananas, and eats K bananas from that pile.  If the pile
# has less than K bananas, she eats all of them instead, and won't eat any more
# bananas during this hour.
#
# Koko likes to eat slowly, but still wants to finish eating all the bananas
# before the guards come back.
#
# Return the minimum integer K such that she can eat all the bananas within H
# hours.
#
#
# Example 1:
# Input: piles = [3,6,7,11], H = 8
# Output: 4
# Example 2:
# Input: piles = [30,11,23,4,20], H = 5
# Output: 30
# Example 3:
# Input: piles = [30,11,23,4,20], H = 6
# Output: 23
#
#
# Constraints:
#
#
#       1 <= piles.length <= 10^4
#       piles.length <= H <= 10^9
#       1 <= piles[i] <= 10^9# Time:  O(nlogr)
# Space: O(1)

class Solution(object):
    def minEatingSpeed(self, piles, H):
        """
        :type piles: List[int]
        :type H: int
        :rtype: int
        """
        def possible(piles, H, K):
            return sum((pile-1)//K+1 for pile in piles) <= H

        left, right = 1, max(piles)
        while left <= right:
            mid = left + (right-left)//2
            if possible(piles, H, mid):
                right = mid-1
            else:
                left = mid+1
        return left
```

random-pick-with-weight.py

```
# Given an array of positive integers w. where w[i] describes the weight of
# ith index (0-indexed).
#
# We need to call the function pickIndex() which randomly returns an integer in
# the range [0, w.length - 1]. pickIndex() should return the integer proportional
# to its weight in the w array. For example, for w = [1, 3], the probability of
# picking index 0 is 1 / (1 + 3) = 0.25 (i.e 25%) while the probability of picking
# index 1 is 3 / (1 + 3) = 0.75 (i.e 75%).
#
# More formally, the probability of picking index i is w[i] / sum(w).
#
#
# Example 1:
#
# Input
# ["Solution","pickIndex"]
# [[[1]],[]]
# Output
# [null,0]
#
# Explanation
# Solution solution = new Solution([1]);
# solution.pickIndex(); // return 0. Since there is only one single element on
# the array the only option is to return the first element.
#
#
# Example 2:
#
# Input
# ["Solution","pickIndex","pickIndex","pickIndex","pickIndex","pickIndex"]
# [[[1,3]],[],[],[],[],[]]
# Output
# [null,1,1,1,1,0]
#
# Explanation
# Solution solution = new Solution([1, 3]);
# solution.pickIndex(); // return 1. It's returning the second element (index =
# 1) that has probability of 3/4.
# solution.pickIndex(); // return 1
# solution.pickIndex(); // return 1
# solution.pickIndex(); // return 1
# solution.pickIndex(); // return 0. It's returning the first element (index =
# 0) that has probability of 1/4.
#
# Since this is a randomization problem, multiple answers are allowed so the
# following outputs can be considered correct :
# [null,1,1,1,1,0]
# [null,1,1,1,1,1]
# [null,1,1,1,0,0]
# [null,1,1,1,0,1]
# [null,1,0,1,0,0]
# ......
# and so on.
#
#
#
# Constraints:
#
```

```python
#
#       1 <= w.length <= 10000
#       1 <= w[i] <= 10^5
#       pickIndex will be called at most 10000 times.# Time:  ctor: O(n)
#        pickIndex: O(logn)
# Space: O(n)

import random
import bisect


class Solution(object):

    def __init__(self, w):
        """
        :type w: List[int]
        """
        self.__prefix_sum = list(w)
        for i in xrange(1, len(w)):
            self.__prefix_sum[i] += self.__prefix_sum[i-1]

    def pickIndex(self):
        """
        :rtype: int
        """
        target = random.randint(0, self.__prefix_sum[-1]-1)
        return bisect.bisect_right(self.__prefix_sum, target)
```

find-k-closest-elements.py

```
# Given a sorted array arr, two integers k and x, find the k closest elements to
# x in the array. The result should also be sorted in ascending order. If there is
# a tie, the smaller elements are always preferred.
#
#
# Example 1:
# Input: arr = [1,2,3,4,5], k = 4, x = 3
# Output: [1,2,3,4]
# Example 2:
# Input: arr = [1,2,3,4,5], k = 4, x = -1
# Output: [1,2,3,4]
#
#
# Constraints:
#
#
#       1 <= k <= arr.length
#       1 <= arr.length <= 10^4
#       Absolute value of elements in the array and x will not exceed 104# Time:  O(logn + k)
# Space: O(1)

import bisect


class Solution(object):
    def findClosestElements(self, arr, k, x):
        """
        :type arr: List[int]
        :type k: int
        :type x: int
        :rtype: List[int]
        """
        i = bisect.bisect_left(arr, x)
        left, right = i-1, i
        while k:
            if right >= len(arr) or \
               (left >= 0 and abs(arr[left]-x) <= abs(arr[right]-x)):
                left -= 1
            else:
                right += 1
            k -= 1
        return arr[left+1:right]
```

jump-game-iii.py

```python
# Given an array of non-negative integers arr, you are initially positioned at
# start index of the array. When you are at index i, you can jump to i + arr[i] or
# i - arr[i], check if you can reach to any index with value 0.
#
# Notice that you can not jump outside of the array at any time.
#
#
# Example 1:
#
# Input: arr = [4,2,3,0,3,1,2], start = 5
# Output: true
# Explanation:
# All possible ways to reach at index 3 with value 0 are:
# index 5 -> index 4 -> index 1 -> index 3
# index 5 -> index 6 -> index 4 -> index 1 -> index 3
#
#
# Example 2:
#
# Input: arr = [4,2,3,0,3,1,2], start = 0
# Output: true
# Explanation:
# One possible way to reach at index 3 with value 0 is:
# index 0 -> index 4 -> index 1 -> index 3
#
#
# Example 3:
#
# Input: arr = [3,0,2,1,2], start = 2
# Output: false
# Explanation: There is no way to reach at index 1 with value 0.
#
#
#
# Constraints:
#
#
#       1 <= arr.length <= 5 * 10^4
#       0 <= arr[i] < arr.length
#       0 <= start < arr.length# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def canReach(self, arr, start):
        """
        :type arr: List[int]
        :type start: int
        :rtype: bool
        """
        q, lookup = collections.deque([start]), set([start])
        while q:
            i = q.popleft()
            if not arr[i]:
                return True
            for j in [i-arr[i], i+arr[i]]:
```

```
        if 0 <= j < len(arr) and j not in lookup:
            lookup.add(j)
            q.append(j)
return False
```

single-number-ii.py

```python
# Given a non-empty array of integers, every element appears three times except
# for one, which appears exactly once. Find that single one.
#
# Note:
#
# Your algorithm should have a linear runtime complexity. Could you implement it
# without using extra memory?
#
# Example 1:
#
# Input: [2,2,3,2]
# Output: 3
#
#
# Example 2:
#
# Input: [0,1,0,1,0,1,99]
# Output: 99# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    # @param A, a list of integer
    # @return an integer
    def singleNumber(self, A):
        one, two = 0, 0
        for x in A:
            one, two = (~x & one) | (x & ~one & ~two), (~x & two) | (x & one)
        return one


class Solution2(object):
    # @param A, a list of integer
    # @return an integer
    def singleNumber(self, A):
        one, two, carry = 0, 0, 0
        for x in A:
            two |= one & x
            one ^= x
            carry = one & two
            one &= ~carry
            two &= ~carry
        return one


class Solution3(object):
    def singleNumber(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        return (collections.Counter(list(set(nums)) * 3) - collections.Counter(nums)).keys()[0]


class Solution4(object):
    def singleNumber(self, nums):
```

```python
        """
        :type nums: List[int]
        :rtype: int
        """
        return (sum(set(nums)) * 3 - sum(nums)) / 2


#  every element appears 4 times except for one with 2 times
class SolutionEX(object):
    # @param A, a list of integer
    # @return an integer
    # [1, 1, 1, 1, 2, 2, 2, 2, 3, 3]
    def singleNumber(self, A):
        one, two, three = 0, 0, 0
        for x in A:
            one, two, three = (~x & one) | (x & ~one & ~two & ~three), (~x & two) | (x & one), (~x & three) |
        return two
```

array-of-doubled-pairs.py

```python
# Example 1:
#
# Input: [3,1,3,6]
# Output: false
#
#
#
# Example 2:
#
# Input: [2,1,2,6]
# Output: false
#
#
#
# Example 3:
#
# Input: [4,-2,2,-4]
# Output: true
# Explanation: We can take two groups, [-2,-4] and [2,4] to form [-2,-4,2,4] or
# [2,4,-2,-4].
#
#
#
# Example 4:
#
# Input: [1,2,4,16,8,4]
# Output: false
#
#
#
#
# Note:
#
#
#       0 <= A.length <= 30000
#       A.length is even
#       -100000 <= A[i] <= 100000# Time:  O(n + klogk)
# Space: O(k)

import collections


class Solution(object):
    def canReorderDoubled(self, A):
        """
        :type A: List[int]
        :rtype: bool
        """
        count = collections.Counter(A)
        for x in sorted(count, key=abs):
            if count[x] > count[2*x]:
                return False
            count[2*x] -= count[x]
        return True
```

jump-game.py

```python
# 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.
#
#
# Example 1:
#
# Input: nums = [2,3,1,1,4]
# Output: true
# Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.
#
#
# Example 2:
#
# Input: nums = [3,2,1,0,4]
# Output: false
# Explanation: You will always arrive at index 3 no matter what. Its maximum
# jump length is 0, which makes it impossible to reach the last index.
#
#
#
# Constraints:
#
#
#       1 <= nums.length <= 3 * 10^4
#       0 <= nums[i][j] <= 10^5# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param A, a list of integers
    # @return a boolean
    def canJump(self, A):
        reachable = 0
        for i, length in enumerate(A):
            if i > reachable:
                break
            reachable = max(reachable, i + length)
        return reachable >= len(A) - 1
```

remove-k-digits.py

```python
# Given a non-negative integer num represented as a string, remove k digits from
# the number so that the new number is the smallest possible.
#
#
# Note:
#
#
# The length of num is less than 10002 and will be  k.
# The given num does not contain any leading zero.
#
#
#
#
# Example 1:
# Input: num = "1432219", k = 3
# Output: "1219"
# Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219
# which is the smallest.
#
#
#
# Example 2:
# Input: num = "10200", k = 1
# Output: "200"
# Explanation: Remove the leading 1 and the number is 200. Note that the output
# must not contain leading zeroes.
#
#
#
# Example 3:
# Input: num = "10", k = 2
# Output: "0"
# Explanation: Remove all the digits from the number and it is left with nothing
# which is 0.# Time:  O(n)
# Space: O(n)

class Solution(object):
    def removeKdigits(self, num, k):
        """
        :type num: str
        :type k: int
        :rtype: str
        """
        result = []
        for d in num:
            while k and result and result[-1] > d:
                result.pop()
                k -= 1
            result.append(d)
        return ''.join(result).lstrip('0')[:-k or None] or '0'
```

construct-quad-tree.py

```
# Given a n * n matrix grid of 0's and 1's only. We want to represent the grid
# with a Quad-Tree.
#
# Return the root of the Quad-Tree representing the grid.
#
# Notice that you can assign the value of a node to True or False when isLeaf is
# False, and both are accepted in the answer.
#
# A Quad-Tree is a tree data structure in which each internal node has exactly
# four children. Besides, each node has two attributes:
#
#
#       val: True if the node represents a grid of 1's or False if the node
# represents a grid of 0's.
#       isLeaf: True if the node is leaf node on the tree or False if the node
# has the four children.
#
#
# class Node {
#     public boolean val;
#     public boolean isLeaf;
#     public Node topLeft;
#     public Node topRight;
#     public Node bottomLeft;
#     public Node bottomRight;
# }
#
# We can construct a Quad-Tree from a two-dimensional area using the following
# steps:
#
#
#       If the current grid has the same value (i.e all 1's or all 0's) set
# isLeaf True and set val to the value of the grid and set the four children to
# Null and stop.
#       If the current grid has different values, set isLeaf to False and set
# val to any value and divide the current grid into four sub-grids as shown in the
# photo.
#       Recurse for each of the children with the proper sub-grid.
#
#
# If you want to know more about the Quad-Tree, you can refer to the wiki.
#
# Quad-Tree format:
#
# The output represents the serialized format of a Quad-Tree using level order
# traversal, where null signifies a path terminator where no node exists below.
#
# It is very similar to the serialization of the binary tree. The only
# difference is that the node is represented as a list [isLeaf, val].
#
# If the value of isLeaf or val is True we represent it as 1 in the
# list [isLeaf, val] and if the value of isLeaf or val is False we represent it as
# 0.
#
#
# Example 1:
#
# Input: grid = [[0,1],[1,0]]
```

```python
# Output: [[0,1],[1,0],[1,1],[1,1],[1,0]]
# Explanation: The explanation of this example is shown below:
# Notice that 0 represnts False and 1 represents True in the photo representing
# the Quad-Tree.
#
#
#
# Example 2:
#
#
#
# Input: grid = [[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,0,0],[1,1,1,1,1,1,1,1],[1,1,1,1,
# 1,1,1,1],[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,0,0]
# ]
# Output:
# [[0,1],[1,1],[0,1],[1,1],[1,0],null,null,null,null,[1,0],[1,0],[1,1],[1,1]]
# Explanation: All values in the grid are not the same. We divide the grid into
# four sub-grids.
# The topLeft, bottomLeft and bottomRight each has the same value.
# The topRight have different values so we divide it into 4 sub-grids where each
# has the same value.
# Explanation is shown in the photo below:
#
#
#
# Example 3:
#
# Input: grid = [[1,1],[1,1]]
# Output: [[1,1]]
#
#
# Example 4:
#
# Input: grid = [[0]]
# Output: [[1,0]]
#
#
# Example 5:
#
# Input: grid = [[1,1,0,0],[1,1,0,0],[0,0,1,1],[0,0,1,1]]
# Output: [[0,1],[1,1],[1,0],[1,0],[1,1]]
#
#
#
# Constraints:
#
#
#       n == grid.length == grid[i].length
#       n == 2^x where 0 <= x <= 6# Time:  O(n)
# Space: O(h)

class Node(object):
    def __init__(self, val, isLeaf, topLeft, topRight, bottomLeft, bottomRight):
        self.val = val
        self.isLeaf = isLeaf
        self.topLeft = topLeft
        self.topRight = topRight
        self.bottomLeft = bottomLeft
        self.bottomRight = bottomRight
```

```python
class Solution(object):
    def construct(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: Node
        """
        def dfs(grid, x, y, l):
            if l == 1:
                 return Node(grid[x][y] == 1, True, None, None, None, None)
            half = l // 2
            topLeftNode = dfs(grid, x, y, half)
            topRightNode = dfs(grid, x, y+half, half)
            bottomLeftNode = dfs(grid, x+half, y, half)
            bottomRightNode = dfs(grid, x+half, y+half, half)
            if topLeftNode.isLeaf and topRightNode.isLeaf and \
               bottomLeftNode.isLeaf and bottomRightNode.isLeaf and \
               topLeftNode.val == topRightNode.val == bottomLeftNode.val == bottomRightNode.val:
                 return Node(topLeftNode.val, True, None, None, None, None)
            return Node(True, False, topLeftNode, topRightNode, bottomLeftNode, bottomRightNode)

        if not grid:
            return None
        return dfs(grid, 0, 0, len(grid))
```

search-in-rotated-sorted-array-ii.py

```python
# Suppose an array sorted in ascending order is rotated at some pivot unknown to
# you beforehand.
#
# (i.e., [0,0,1,2,2,5,6] might become [2,5,6,0,0,1,2]).
#
# You are given a target value to search. If found in the array return true,
# otherwise return false.
#
# Example 1:
#
# Input: nums = [2,5,6,0,0,1,2], target = 0
# Output: true
#
#
# Example 2:
#
# Input: nums = [2,5,6,0,0,1,2], target = 3
# Output: false
#
# Follow up:
#
#
#       This is a follow up problem to Search in Rotated Sorted Array, where
# nums may contain duplicates.
#       Would this affect the run-time complexity? How and why?# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        left, right = 0, len(nums) - 1

        while left <= right:
            mid = left + (right - left) / 2

            if nums[mid] == target:
                return True
            elif nums[mid] == nums[left]:
                left += 1
            elif (nums[mid] > nums[left] and nums[left] <= target < nums[mid]) or \
                 (nums[mid] < nums[left] and not (nums[mid] < target <= nums[right])):
                right = mid - 1
            else:
                left = mid + 1

        return False
```

delete-operation-for-two-strings.py

```
# Given two words word1 and word2, find the minimum number of steps required to
# make word1 and word2 the same, where in each step you can delete one character
# in either string.
#
#
# Example 1:
#
# Input: "sea", "eat"
# Output: 2
# Explanation: You need one step to make "sea" to "ea" and another step to make
# "eat" to "ea".
#
#
#
# Note:
#
#
# The length of given words won't exceed 500.
# Characters in given words can only be lower-case letters.# Time:  O(m * n)
# Space: O(n)

class Solution(object):
    def minDistance(self, word1, word2):
        """
        :type word1: str
        :type word2: str
        :rtype: int
        """
        m, n = len(word1), len(word2)
        dp = [[0] * (n+1) for _ in range(2)]
        for i in range(m):
            for j in range(n):
                dp[(i+1)%2][j+1] = max(dp[i%2][j+1], \
                                       dp[(i+1)%2][j], \
                                       dp[i%2][j] + (word1[i] == word2[j]))
        print(dp)
        return m + n - 2*dp[m%2][n]

    def minDistance2(self, word1, word2):
        """
        :type word1: str
        :type word2: str
        :rtype: int
        """
        m, n = len(word1), len(word2)
        dp = [[0] * (3) for _ in range(2)]
        for i in range(m):
            for j in range(n):
                dp[(i+1)%2][(j+1)%2] = max(dp[i%2][(j+1)%2], \
                                       dp[(i+1)%2][j%2], \
                                       dp[i%2][j%2] + (word1[i] == word2[j]))
        print(dp)
        return m + n - 2*dp[m%2][n]

if __name__ == '__main__':
    s, t = "mart", "karma"
    ret = Solution().minDistance(s, t)
    ret = Solution().minDistance2(s, t)
```

```
print(ret)
```

length-of-longest-fibonacci-subsequence.py

```python
# A sequence X_1, X_2, ..., X_n is fibonacci-like if:
#
#
#       n >= 3
#       X_i + X_{i+1} = X_{i+2} for all i + 2 <= n
#
#
# Given a strictly increasing array A of positive integers forming a sequence,
# find the length of the longest fibonacci-like subsequence of A.  If one does not
# exist, return 0.
#
# (Recall that a subsequence is derived from another sequence A by deleting any
# number of elements (including none) from A, without changing the order of the
# remaining elements.  For example, [3, 5, 8] is a subsequence of [3, 4, 5, 6, 7,
# 8].)
#
#
#
#
#
#
# Example 1:
#
# Input: [1,2,3,4,5,6,7,8]
# Output: 5
# Explanation:
# The longest subsequence that is fibonacci-like: [1,2,3,5,8].
#
#
# Example 2:
#
# Input: [1,3,7,11,12,14,18]
# Output: 3
# Explanation:
# The longest subsequence that is fibonacci-like:
# [1,11,12], [3,11,14] or [7,11,18].
#
#
#
#
# Note:
#
#
#       3 <= A.length <= 1000
#       1 <= A[0] < A[1] < ... < A[A.length - 1] <= 10^9
#       (The time limit has been reduced by 50% for submissions in Java, C, and
# C++.)# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def lenLongestFibSubseq(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        lookup = set(A)
        result = 2
        for i in xrange(len(A)):
```

```
    for j in xrange(i+1, len(A)):
        x, y, l = A[i], A[j], 2
        while x+y in lookup:
            x, y, l = y, x+y, l+1
        result = max(result, l)
return result if result > 2 else 0
```

bitwise-ors-of-subarrays.py

```python
# Example 1:
#
# Input: [0]
# Output: 1
# Explanation:
# There is only one possible result: 0.
#
#
#
# Example 2:
#
# Input: [1,1,2]
# Output: 3
# Explanation:
# The possible subarrays are [1], [1], [2], [1, 1], [1, 2], [1, 1, 2].
# These yield the results 1, 1, 2, 1, 3, 3.
# There are 3 unique values, so the answer is 3.
#
#
#
# Example 3:
#
# Input: [1,2,4]
# Output: 6
# Explanation:
# The possible results are 1, 2, 3, 4, 6, and 7.# Time:  O(32 * n)
# Space: O(1)

class Solution(object):
    def subarrayBitwiseORs(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        result, curr = set(), {0}
        for i in A:
            curr = {i} | {i | j for j in curr}
            result |= curr
        return len(result)
```

check-completeness-of-a-binary-tree.py

```python
# Example 2:
#
#
#
# Input: [1,2,3,4,5,null,7]
# Output: false
# Explanation: The node with value 7 isn't as far left as possible.# Time:  O(n)
# Space: O(w)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def isCompleteTree(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        end = False
        current = [root]
        while current:
            next_level = []
            for node in current:
                if not node:
                    end = True
                    continue
                if end:
                    return False
                next_level.append(node.left)
                next_level.append(node.right)
            current = next_level
        return  True


# Time:  O(n)
# Space: O(w)
class Solution2(object):
    def isCompleteTree(self, root):
        """
        :type root: TreeNode
        :rtype: bool
        """
        prev_level, current = [], [(root, 1)]
        count = 0
        while current:
            count += len(current)
            next_level = []
            for node, v in current:
                if not node:
                    continue
                next_level.append((node.left, 2*v))
                next_level.append((node.right, 2*v+1))
            prev_level, current = current, next_level
```

```
return prev_level[-1][1] == count
```

increasing-subsequences.py

```python
# Given an integer array, your task is to find all the different possible
# increasing subsequences of the given array, and the length of an increasing
# subsequence should be at least 2.
#
#
#
# Example:
#
# Input: [4, 6, 7, 7]
# Output: [[4, 6], [4, 7], [4, 6, 7], [4, 6, 7, 7], [6, 7], [6, 7, 7], [7,7],
# [4,7,7]]
#
#
#
# Constraints:
#
#
#       The length of the given array will not exceed 15.
#       The range of integer in the given array is [-100,100].
#       The given array may contain duplicates, and two equal integers should
# also be considered as a special case of increasing sequence.# Time:  O(n * 2^n)
# Space: O(n), longest possible path in tree, which is if all numbers are increasing.

class Solution(object):
    def findSubsequences(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        def findSubsequencesHelper(nums, pos, seq, result):
            if len(seq) >= 2:
                result.append(list(seq))
            lookup = set()
            for i in xrange(pos, len(nums)):
                if (not seq or nums[i] >= seq[-1]) and \
                   nums[i] not in lookup:
                    lookup.add(nums[i])
                    seq.append(nums[i])
                    findSubsequencesHelper(nums, i+1, seq, result)
                    seq.pop()

        result, seq = [], []
        findSubsequencesHelper(nums, 0, seq, result)
        return result
```

binary-string-with-substrings-representing-1-to-n.py

```python
# Given a binary string S (a string consisting only of '0' and '1's) and a
# positive integer N, return true if and only if for every integer X from 1 to N,
# the binary representation of X is a substring of S.
#
#
#
# Example 1:
#
# Input: S = "0110", N = 3
# Output: true
#
#
# Example 2:
#
# Input: S = "0110", N = 4
# Output: false
#
#
#
#
# Note:
#
#
#       1 <= S.length <= 1000
#       1 <= N <= 10^9# Time:  O(n^2), n is the length of S
# Space: O(1)

class Solution(object):
    def queryString(self, S, N):
        """
        :type S: str
        :type N: int
        :rtype: bool
        """
        # since S with length n has at most different n-k+1 k-digit numbers
        # => given S with length n, valid N is at most 2(n-k+1)
        # => valid N <= 2(n-k+1) < 2n = 2 * S.length
        return all(bin(i)[2:] in S for i in reversed(xrange(N//2, N+1)))
```

valid-parenthesis-string.py

```python
# Given a string containing only three types of characters: '(', ')' and '*',
# write a function to check whether this string is valid. We define the validity
# of a string by these rules:
#
# Any left parenthesis '(' must have a corresponding right parenthesis ')'.
# Any right parenthesis ')' must have a corresponding left parenthesis '('.
# Left parenthesis '(' must go before the corresponding right parenthesis ')'.
# '*' could be treated as a single right parenthesis ')' or a single left
# parenthesis '(' or an empty string.
# An empty string is also valid.
#
#
#
# Example 1:
#
# Input: "()"
# Output: True
#
#
#
# Example 2:
#
# Input: "(*)"
# Output: True
#
#
#
# Example 3:
#
# Input: "(*))"
# Output: True
#
#
#
# Note:
#
#
# The string size will be in the range [1, 100].# Time:  O(n)
# Space: O(1)

class Solution(object):
    def checkValidString(self, s):
        """
        :type s: str
        :rtype: bool
        """
        lower, upper = 0, 0  # keep lower bound and upper bound of '(' counts
        for c in s:
            lower += 1 if c == '(' else -1
            upper -= 1 if c == ')' else -1
            if upper < 0: break
            lower = max(lower, 0)
        return lower == 0  # range of '(' count is valid
```

total-hamming-distance.py

```
# The Hamming distance between two integers is the number of positions at which
# the corresponding bits are different.
#
# Now your job is to find the total Hamming distance between all pairs of the
# given numbers.
#
#
# Example:
#
# Input: 4, 14, 2
#
# Output: 6
#
# Explanation: In binary representation, the 4 is 0100, 14 is 1110, and 2 is
# 0010 (just
# showing the four bits relevant in this case). So the answer will be:
# HammingDistance(4, 14) + HammingDistance(4, 2) + HammingDistance(14, 2) = 2 +
# 2 + 2 = 6.
#
#
#
# Note:
#
#
# Elements of the given array are in the range of 0  to 10^9
# Length of the array will not exceed 10^4.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def totalHammingDistance(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result = 0
        for i in xrange(32):
            counts = [0] * 2
            for num in nums:
                counts[(num >> i) & 1] += 1
            result += counts[0] * counts[1]
        return result
```

serialize-and-deserialize-bst.py

```python
# 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 search tree. There
# is no restriction on how your serialization/deserialization algorithm should
# work. You just need to ensure that a binary search tree can be serialized to a
# string and this string can be deserialized to the original tree structure.
#
# The encoded string should be as compact as possible.
#
# Note: Do not use class member/global/static variables to store states. Your
# serialize and deserialize algorithms should be stateless.# Time:  O(n)
# Space: O(h)

import collections


class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Codec(object):

    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        def serializeHelper(node, vals):
            if node:
                vals.append(node.val)
                serializeHelper(node.left, vals)
                serializeHelper(node.right, vals)

        vals = []
        serializeHelper(root, vals)

        return ' '.join(map(str, vals))


    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """
        def deserializeHelper(minVal, maxVal, vals):
            if not vals:
                return None

            if minVal < vals[0] < maxVal:
                val = vals.popleft()
```

```python
            node = TreeNode(val)
            node.left = deserializeHelper(minVal, val, vals)
            node.right = deserializeHelper(val, maxVal, vals)
            return node
        else:
            return None

    vals = collections.deque([int(val) for val in data.split()])

    return deserializeHelper(float('-inf'), float('inf'), vals)
```

product-of-array-except-self.py

```
# Given an array nums of n integers where n > 1,  return an array output such
# that output[i] is equal to the product of all the elements of nums except
# nums[i].
#
# Example:
#
# Input:  [1,2,3,4]
# Output: [24,12,8,6]
#
#
# Constraint: It's guaranteed that the product of the elements of any prefix or
# suffix of the array (including the whole array) fits in a 32 bit integer.
#
# Note: Please solve it without division and in O(n).
#
# Follow up:
#
# Could you solve it with constant space complexity? (The output array does not
# count as extra space for the purpose of space complexity analysis.)# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param {integer[]} nums
    # @return {integer[]}
    def productExceptSelf(self, nums):
        if not nums:
            return []

        left_product = [1 for _ in xrange(len(nums))]
        for i in xrange(1, len(nums)):
            left_product[i] = left_product[i - 1] * nums[i - 1]

        right_product = 1
        for i in xrange(len(nums) - 2, -1, -1):
            right_product *= nums[i + 1]
            left_product[i] = left_product[i] * right_product

        return left_product
```

queue-reconstruction-by-height.py

```python
# Suppose you have a random list of people standing in a queue. Each person is
# described by a pair of integers (h, k), where h is the height of the person and
# k is the number of people in front of this person who have a height greater than
# or equal to h. Write an algorithm to reconstruct the queue.
#
# Note:
#
# The number of people is less than 1,100.
#
#
# Example
#
# Input:
# [[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]
#
# Output:
# [[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]# Time:  O(n * sqrt(n))
# Space: O(n)

class Solution(object):
    def reconstructQueue(self, people):
        """
        :type people: List[List[int]]
        :rtype: List[List[int]]
        """
        people.sort(key=lambda h_k: (-h_k[0], h_k[1]))

        blocks = [[]]
        for p in people:
            index = p[1]

            for i, block in enumerate(blocks):
                if index <= len(block):
                    break
                index -= len(block)
            block.insert(index, p)

            if len(block) * len(block) > len(people):
                blocks.insert(i+1, block[len(block)/2:])
                del block[len(block)/2:]

        return [p for block in blocks for p in block]


# Time:  O(n^2)
# Space: O(n)
class Solution2(object):
    def reconstructQueue(self, people):
        """
        :type people: List[List[int]]
        :rtype: List[List[int]]
        """
        people.sort(key=lambda h_k1: (-h_k1[0], h_k1[1]))
        result = []
        for p in people:
            result.insert(p[1], p)
        return result
```

building-h2o.py

```
# There are two kinds of threads, oxygen and hydrogen. Your goal is to group
# these threads to form water molecules. There is a barrier where each thread has
# to wait until a complete molecule can be formed. Hydrogen and oxygen threads
# will be given releaseHydrogen and releaseOxygen methods respectively, which will
# allow them to pass the barrier. These threads should pass the barrier in groups
# of three, and they must be able to immediately bond with each other to form a
# water molecule. You must guarantee that all the threads from one molecule bond
# before any other threads from the next molecule do.
#
# In other words:
#
#
#       If an oxygen thread arrives at the barrier when no hydrogen threads are
# present, it has to wait for two hydrogen threads.
#       If a hydrogen thread arrives at the barrier when no other threads are
# present, it has to wait for an oxygen thread and another hydrogen thread.
#
#
# We don't have to worry about matching the threads up explicitly; that is, the
# threads do not necessarily know which other threads they are paired up with. The
# key is just that threads pass the barrier in complete sets; thus, if we examine
# the sequence of threads that bond and divide them into groups of three, each
# group should contain one oxygen and two hydrogen threads.
#
# Write synchronization code for oxygen and hydrogen molecules that enforces
# these constraints.
#
#
#
#
#
#
# Example 1:
#
# Input: "HOH"
# Output: "HHO"
# Explanation: "HOH" and "OHH" are also valid answers.
#
#
#
# Example 2:
#
# Input: "OOHHHH"
# Output: "HHOHHO"
# Explanation: "HOHHHO", "OHHHHO", "HHOHOH", "HOHHOH", "OHHHOH", "HHOOHH",
# "HOHOHH" and "OHHOHH" are also valid answers.
#
#
#
#
#
# Constraints:
#
#
#       Total length of input string will be 3n, where 1  n  20.
#       Total number of H will be 2n in the input string.
#       Total number of O will be n in the input string.# Time:  O(n)
# Space: O(1)
```

```python
import threading


class H2O(object):
    def __init__(self):
        self.__l = threading.Lock()
        self.__nH = 0
        self.__nO = 0
        self.__releaseHydrogen = None
        self.__releaseOxygen = None

    def hydrogen(self, releaseHydrogen):
        with self.__l:
            self.__releaseHydrogen = releaseHydrogen
            self.__nH += 1
            self.__output()

    def oxygen(self, releaseOxygen):
        with self.__l:
            self.__releaseOxygen = releaseOxygen
            self.__nO += 1
            self.__output()

    def __output(self):
        while self.__nH >= 2 and \
              self.__nO >= 1:
            self.__nH -= 2
            self.__nO -= 1
            self.__releaseHydrogen()
            self.__releaseHydrogen()
            self.__releaseOxygen()


# Time:  O(n)
# Space: O(1)
# TLE
class H2O2(object):
    def __init__(self):
        self.__nH = 0
        self.__nO = 0
        self.__cv = threading.Condition()

    def hydrogen(self, releaseHydrogen):
        """
        :type releaseHydrogen: method
        :rtype: void
        """
        with self.__cv:
            while (self.__nH+1) - 2*self.__nO > 2:
                self.__cv.wait()
            self.__nH += 1
            # releaseHydrogen() outputs "H". Do not change or remove this line.
            releaseHydrogen()
            self.__cv.notifyAll()

    def oxygen(self, releaseOxygen):
        """
        :type releaseOxygen: method
        :rtype: void
```

```python
"""
with self.__cv:
    while 2*(self.__nO+1) - self.__nH > 2:
        self.__cv.wait()
    self.__nO += 1
    # releaseOxygen() outputs "O". Do not change or remove this line.
    releaseOxygen()
    self.__cv.notifyAll()
```

next-greater-element-ii.py

```
# Given a circular array (the next element of the last element is the first
# element of the array), print the Next Greater Number for every element. The Next
# Greater Number of a number x is the first greater number to its traversing-order
# next in the array, which means you could search circularly to find its next
# greater number. If it doesn't exist, output -1 for this number.
#
#
# Example 1:
#
# Input: [1,2,1]
# Output: [2,-1,2]
# Explanation: The first 1's next greater number is 2;
# The number 2 can't find next greater number;
# The second 1's next greater number needs to search circularly, which is also
# 2.
#
#
#
# Note:
# The length of given array won't exceed 10000.# Time:  O(n)
# Space: O(n)

class Solution(object):
    def nextGreaterElements(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        result, stk = [0] * len(nums), []
        for i in reversed(xrange(2*len(nums))):
            while stk and stk[-1] <= nums[i % len(nums)]:
                stk.pop()
            result[i % len(nums)] = stk[-1] if stk else -1
            stk.append(nums[i % len(nums)])
        return result
```

rotate-image.py

```python
# You are given an n x n 2D matrix representing an image, rotate the image by 90
# degrees (clockwise).
#
# You have to rotate the image in-place, which means you have to modify the
# input 2D matrix directly. DO NOT allocate another 2D matrix and do the rotation.
#
#
# Example 1:
#
# Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
# Output: [[7,4,1],[8,5,2],[9,6,3]]
#
#
# Example 2:
#
# Input: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]
# Output: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]
#
#
# Example 3:
#
# Input: matrix = [[1]]
# Output: [[1]]
#
#
# Example 4:
#
# Input: matrix = [[1,2],[3,4]]
# Output: [[3,1],[4,2]]
#
#
#
# Constraints:
#
#
#       matrix.length == n
#       matrix[i].length == n
#       1 <= n <= 20
#       -1000 <= matrix[i][j] <= 1000# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    # @param matrix, a list of lists of integers
    # @return a list of lists of integers
    def rotate(self, matrix):
        n = len(matrix)

        # anti-diagonal mirror
        for i in xrange(n):
            for j in xrange(n - i):
                matrix[i][j], matrix[n-1-j][n-1-i] = matrix[n-1-j][n-1-i], matrix[i][j]

        # horizontal mirror
        for i in xrange(n / 2):
            for j in xrange(n):
                matrix[i][j], matrix[n-1-i][j] = matrix[n-1-i][j], matrix[i][j]

        return matrix
```

```python
# Time:  O(n^2)
# Space: O(n^2)
class Solution2(object):
    # @param matrix, a list of lists of integers
    # @return a list of lists of integers
    def rotate(self, matrix):
        return [list(reversed(x)) for x in zip(*matrix)]
```

surrounded-regions.py

```python
# Given a 2D board containing 'X' and 'O' (the letter O), capture all regions
# surrounded by 'X'.
#
# A region is captured by flipping all 'O's into 'X's in that surrounded region.
#
# Example:
#
# X X X X
# X O O X
# X X O X
# X O X X
#
#
# After running your function, the board should be:
#
# X X X X
# X X X X
# X X X X
# X O X X
#
#
# Explanation:
#
# Surrounded regions shouldn't be on the border, which means that any 'O' on the
# border of the board are not flipped to 'X'. Any 'O' that is not on the border
# and it is not connected to an 'O' on the border will be flipped to 'X'. Two
# cells are connected if they are adjacent cells connected horizontally or
# vertically.# Time:  O(m * n)
# Space: O(m + n)

import collections


class Solution(object):
    def solve(self, board):
        """
        :type board: List[List[str]]
        :rtype: void Do not return anything, modify board in-place instead.
        """
        if not board:
            return

        q = collections.deque()

        for i in xrange(len(board)):
            if board[i][0] == 'O':
                board[i][0] = 'V'
                q.append((i, 0))
            if board[i][len(board[0])-1] == 'O':
                board[i][len(board[0])-1] = 'V'
                q.append((i, len(board[0])-1))

        for j in xrange(1, len(board[0])-1):
            if board[0][j] == 'O':
                board[0][j] = 'V'
                q.append((0, j))
            if board[len(board)-1][j] == 'O':
                board[len(board)-1][j] = 'V'
```

```python
        q.append((len(board)-1, j))

    while q:
        i, j = q.popleft()
        for x, y in [(i+1, j), (i-1, j), (i, j+1), (i, j-1)]:
            if 0 <= x < len(board) and 0 <= y < len(board[0]) and \
               board[x][y] == 'O':
                board[x][y] = 'V'
                q.append((x, y))

    for i in xrange(len(board)):
        for j in xrange(len(board[0])):
            if board[i][j] != 'V':
                board[i][j] = 'X'
            else:
                board[i][j] = 'O'
```

magical-string.py

```python
# A magical string S consists of only '1' and '2' and obeys the following rules:
#
#
# The string S is magical because concatenating the number of contiguous
# occurrences of characters '1' and '2' generates the string S itself.
#
#
#
# The first few elements of string S is the following:
# S = "1221121221221121122......"
#
#
#
# If we group the consecutive '1's and '2's in S, it will be:
#
#
# 1   22  11  2  1  22  1  22  11  2  11  22 ......
#
#
# and the occurrences of '1's or '2's in each group are:
#
#
# 1   2    2    1   1    2     1    2     2    1    2    2 ......
#
#
#
# You can see that the occurrence sequence above is the S itself.
#
#
#
# Given an integer N as input, return the number of '1's in the first N number
# in the magical string S.
#
#
# Note:
# N will not exceed 100,000.
#
#
#
# Example 1:
#
# Input: 6
# Output: 3
# Explanation: The first 6 elements of magical string S is "12211" and it
# contains three 1's, so return 3.# Time:  O(n)
# Space: O(logn)

import itertools


class Solution(object):
    def magicalString(self, n):
        """
        :type n: int
        :rtype: int
        """
        def gen():  # see figure 1 on page 3 of http://www.emis.ams.org/journals/JIS/VOL15/Nilsson/nilsson5.pd
            for c in 1, 2, 2:
```

```
            yield c
        for i, c in enumerate(gen()):
            if i > 1:
                for _ in xrange(c):
                    yield i % 2 + 1

    return sum(c & 1 for c in itertools.islice(gen(), n))
```

split-array-into-consecutive-subsequences.py

```python
# Given an array nums sorted in ascending order, return true if and only if you
# can split it into 1 or more subsequences such that each subsequence consists of
# consecutive integers and has length at least 3.
#
#
#
# Example 1:
#
# Input: [1,2,3,3,4,5]
# Output: True
# Explanation:
# You can split them into two consecutive subsequences :
# 1, 2, 3
# 3, 4, 5
#
#
#
# Example 2:
#
# Input: [1,2,3,3,4,4,5,5]
# Output: True
# Explanation:
# You can split them into two consecutive subsequences :
# 1, 2, 3, 4, 5
# 3, 4, 5
#
#
#
# Example 3:
#
# Input: [1,2,3,4,4,5]
# Output: False
#
#
#
#
# Constraints:
#
#
#       1 <= nums.length <= 10000# Time:  O(n)
# Space: O(1)

class Solution(object):
    def isPossible(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        pre, cur = float("-inf"), 0
        cnt1, cnt2, cnt3 = 0, 0, 0
        i = 0
        while i < len(nums):
            cnt = 0
            cur = nums[i]
            while i < len(nums) and cur == nums[i]:
                cnt += 1
                i += 1
```

```
        if cur != pre + 1:
            if cnt1 != 0 or cnt2 != 0:
                return False
            cnt1, cnt2, cnt3 = cnt, 0, 0
        else:
            if cnt < cnt1 + cnt2:
                return False
            cnt1, cnt2, cnt3 = max(0, cnt - (cnt1 + cnt2 + cnt3)), \
                               cnt1, \
                               cnt2 + min(cnt3, cnt - (cnt1 + cnt2))
        pre = cur
    return cnt1 == 0 and cnt2 == 0
```

beautiful-array.py

```python
# Example 2:
#
# Input: 5
# Output: [3,1,2,5,4]# Time:  O(n)
# Space: O(n)

class Solution(object):
    def beautifulArray(self, N):
        """
        :type N: int
        :rtype: List[int]
        """
        result = [1]
        while len(result) < N:
            result = [i*2 - 1 for i in result] + [i*2 for i in result]
        return [i for i in result if i <= N]
```

path-sum-iii.py

```python
# You are given a binary tree in which each node contains an integer value.
#
# Find the number of paths that sum to a given value.
#
# The path does not need to start or end at the root or a leaf, but it must go
# downwards
# (traveling only from parent nodes to child nodes).
#
# The tree has no more than 1,000 nodes and the values are in the range
# -1,000,000 to 1,000,000.
#
# Example:
# root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8
#
#       10
#      /  \
#     5   -3
#    / \    \
#   3   2   11
#  / \   \
# 3  -2   1
#
# Return 3. The paths that sum to 8 are:
#
# 1.  5 -> 3
# 2.  5 -> 2 -> 1
# 3. -3 -> 11# Time:  O(n)
# Space: O(h)

import collections


class Solution(object):
    def pathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: int
        """
        def pathSumHelper(root, curr, sum, lookup):
            if root is None:
                return 0
            curr += root.val
            result = lookup[curr-sum] if curr-sum in lookup else 0
            lookup[curr] += 1
            result += pathSumHelper(root.left, curr, sum, lookup) + \
                      pathSumHelper(root.right, curr, sum, lookup)
            lookup[curr] -= 1
            if lookup[curr] == 0:
                del lookup[curr]
            return result

        lookup = collections.defaultdict(int)
        lookup[0] = 1
        return pathSumHelper(root, 0, sum, lookup)


# Time:  O(n^2)
```

```python
# Space: O(h)
class Solution2(object):
    def pathSum(self, root, sum):
        """
        :type root: TreeNode
        :type sum: int
        :rtype: int
        """
        def pathSumHelper(root, prev, sum):
            if root is None:
                return 0

            curr = prev + root.val
            return int(curr == sum) + \
                   pathSumHelper(root.left, curr, sum) + \
                   pathSumHelper(root.right, curr, sum)

        if root is None:
            return 0

        return pathSumHelper(root, 0, sum) + \
               self.pathSum(root.left, sum) + \
               self.pathSum(root.right, sum)
```

design-underground-system.py

```
# Implement the class UndergroundSystem that supports three methods:
#
# 1. checkIn(int id, string stationName, int t)
#
#
#       A customer with id card equal to id, gets in the station stationName at
# time t.
#       A customer can only be checked into one place at a time.
#
#
# 2. checkOut(int id, string stationName, int t)
#
#
#       A customer with id card equal to id, gets out from the station
# stationName at time t.
#
#
# 3. getAverageTime(string startStation, string endStation)
#
#
#       Returns the average time to travel between the startStation and the
# endStation.
#       The average time is computed from all the previous traveling from
# startStation to endStation that happened directly.
#       Call to getAverageTime is always valid.
#
#
# You can assume all calls to checkIn and checkOut methods are consistent. That
# is, if a customer gets in at time t1 at some station, then it gets out at time
# t2 with t2 > t1. All events happen in chronological order.
#
#
# Example 1:
#
# Input
# ["UndergroundSystem","checkIn","checkIn","checkIn","checkOut","checkOut","chec
# kOut","getAverageTime","getAverageTime","checkIn","getAverageTime","checkOut","g
# etAverageTime"]
# [[],[45,"Leyton",3],[32,"Paradise",8],[27,"Leyton",10],[45,"Waterloo",15],[27,
# "Waterloo",20],[32,"Cambridge",22],["Paradise","Cambridge"],["Leyton","Waterloo"
# ],[10,"Leyton",24],["Leyton","Waterloo"],[10,"Waterloo",38],["Leyton","Waterloo"
# ]]
#
# Output
# [null,null,null,null,null,null,null,14.00000,11.00000,null,11.00000,null,12.00
# 000]
#
# Explanation
# UndergroundSystem undergroundSystem = new UndergroundSystem();
# undergroundSystem.checkIn(45, "Leyton", 3);
# undergroundSystem.checkIn(32, "Paradise", 8);
# undergroundSystem.checkIn(27, "Leyton", 10);
# undergroundSystem.checkOut(45, "Waterloo", 15);
# undergroundSystem.checkOut(27, "Waterloo", 20);
# undergroundSystem.checkOut(32, "Cambridge", 22);
# undergroundSystem.getAverageTime("Paradise", "Cambridge");       // return
# 14.00000. There was only one travel from "Paradise" (at time 8) to "Cambridge"
# (at time 22)
```

```python
# undergroundSystem.getAverageTime("Leyton", "Waterloo");          // return
# 11.00000. There were two travels from "Leyton" to "Waterloo", a customer with
# id=45 from time=3 to time=15 and a customer with id=27 from time=10 to time=20.
# So the average time is ( (15-3) + (20-10) ) / 2 = 11.00000
# undergroundSystem.checkIn(10, "Leyton", 24);
# undergroundSystem.getAverageTime("Leyton", "Waterloo");          // return
# 11.00000
# undergroundSystem.checkOut(10, "Waterloo", 38);
# undergroundSystem.getAverageTime("Leyton", "Waterloo");          // return
# 12.00000
#
#
# Example 2:
#
# Input
# ["UndergroundSystem","checkIn","checkOut","getAverageTime","checkIn","checkOut
# ","getAverageTime","checkIn","checkOut","getAverageTime"]
# [[],[10,"Leyton",3],[10,"Paradise",8],["Leyton","Paradise"],[5,"Leyton",10],[5
# ,"Paradise",16],["Leyton","Paradise"],[2,"Leyton",21],[2,"Paradise",30],["Leyton
# ","Paradise"]]
#
# Output
# [null,null,null,5.00000,null,null,5.50000,null,null,6.66667]
#
# Explanation
# UndergroundSystem undergroundSystem = new UndergroundSystem();
# undergroundSystem.checkIn(10, "Leyton", 3);
# undergroundSystem.checkOut(10, "Paradise", 8);
# undergroundSystem.getAverageTime("Leyton", "Paradise"); // return 5.00000
# undergroundSystem.checkIn(5, "Leyton", 10);
# undergroundSystem.checkOut(5, "Paradise", 16);
# undergroundSystem.getAverageTime("Leyton", "Paradise"); // return 5.50000
# undergroundSystem.checkIn(2, "Leyton", 21);
# undergroundSystem.checkOut(2, "Paradise", 30);
# undergroundSystem.getAverageTime("Leyton", "Paradise"); // return 6.66667
#
#
#
# Constraints:
#
#
#       There will be at most 20000 operations.
#       1 <= id, t <= 10^6
#       All strings consist of uppercase, lowercase English letters and digits.
#       1 <= stationName.length <= 10
#       Answers within 10^-5 of the actual value will be accepted as correct.# Time:  ctor:       O(1)
#        checkin:    O(1)
#        checkout:   O(1)
#        getaverage: O(1)
# Space: O(n)

import collections


class UndergroundSystem(object):

    def __init__(self):
        self.__live = {}
        self.__statistics = collections.defaultdict(lambda: [0, 0])
```

```python
def checkIn(self, id, stationName, t):
    """
    :type id: int
    :type stationName: str
    :type t: int
    :rtype: None
    """
    self.__live[id] = (stationName, t)

def checkOut(self, id, stationName, t):
    """
    :type id: int
    :type stationName: str
    :type t: int
    :rtype: None
    """
    startStation, startTime = self.__live.pop(id)
    self.__statistics[startStation, stationName][0] += t-startTime
    self.__statistics[startStation, stationName][1] += 1

def getAverageTime(self, startStation, endStation):
    """
    :type startStation: str
    :type endStation: str
    :rtype: float
    """
    total_time, cnt = self.__statistics[startStation, endStation]
    return float(total_time) / cnt
```

find-all-duplicates-in-an-array.py

```
# Given an array of integers, 1  a[i]  n (n = size of array), some elements
# appear twice and others appear once.
#
# Find all the elements that appear twice in this array.
#
# Could you do it without extra space and in O(n) runtime?
#
# Example:
#
# Input:
# [4,3,2,7,8,2,3,1]
#
# Output:
# [2,3]# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        result = []
        for i in nums:
            if nums[abs(i)-1] < 0:
                result.append(abs(i))
            else:
                nums[abs(i)-1] *= -1
        return result


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def findDuplicates(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        result = []
        i = 0
        while i < len(nums):
            if nums[i] != nums[nums[i]-1]:
                nums[nums[i]-1], nums[i] = nums[i], nums[nums[i]-1]
            else:
                i += 1

        for i in xrange(len(nums)):
            if i != nums[i]-1:
                result.append(nums[i])
        return result


# Time:  O(n)
# Space: O(n), this doesn't satisfy the question
from collections import Counter
class Solution3(object):
    def findDuplicates(self, nums):
```

```python
"""
:type nums: List[int]
:rtype: List[int]
"""
return [elem for elem, count in Counter(nums).items() if count == 2]
```

zigzag-conversion.py

```python
# The string "PAYPALISHIRING" is written in a zigzag pattern on a given number
# of rows like this: (you may want to display this pattern in a fixed font for
# better legibility)
#
# P   A   H   N
# A P L S I I G
# Y   I   R
#
#
# And then read line by line: "PAHNAPLSIIGYIR"
#
# Write the code that will take a string and make this conversion given a number
# of rows:
#
# string convert(string s, int numRows);
#
# Example 1:
#
# Input: s = "PAYPALISHIRING", numRows = 3
# Output: "PAHNAPLSIIGYIR"
#
#
# Example 2:
#
# Input: s = "PAYPALISHIRING", numRows = 4
# Output: "PINALSIGYAHRPI"
# Explanation:
#
# P     I    N
# A   L S  I G
# Y A   H R
# P     I# Time:  O(n)
# Space: O(1)

class Solution(object):
    def convert(self, s, numRows):
        """
        :type s: str
        :type numRows: int
        :rtype: str
        """
        if numRows == 1:
            return s
        step, zigzag = 2 * numRows - 2, ""
        for i in xrange(numRows):
            for j in xrange(i, len(s), step):
                zigzag += s[j]
                if 0 < i < numRows - 1 and j + step - 2 * i < len(s):
                    zigzag += s[j + step - 2 * i]
        return zigzag
```

number-of-dice-rolls-with-target-sum.py

```python
# You have d dice, and each die has f faces numbered 1, 2, ..., f.
#
# Return the number of possible ways (out of fd total ways) modulo 10^9 + 7 to
# roll the dice so the sum of the face up numbers equals target.
#
#
# Example 1:
#
# Input: d = 1, f = 6, target = 3
# Output: 1
# Explanation:
# You throw one die with 6 faces.  There is only one way to get a sum of 3.
#
#
# Example 2:
#
# Input: d = 2, f = 6, target = 7
# Output: 6
# Explanation:
# You throw two dice, each with 6 faces.  There are 6 ways to get a sum of 7:
# 1+6, 2+5, 3+4, 4+3, 5+2, 6+1.
#
#
# Example 3:
#
# Input: d = 2, f = 5, target = 10
# Output: 1
# Explanation:
# You throw two dice, each with 5 faces.  There is only one way to get a sum of
# 10: 5+5.
#
#
# Example 4:
#
# Input: d = 1, f = 2, target = 3
# Output: 0
# Explanation:
# You throw one die with 2 faces.  There is no way to get a sum of 3.
#
#
# Example 5:
#
# Input: d = 30, f = 30, target = 500
# Output: 222616187
# Explanation:
# The answer must be returned modulo 10^9 + 7.
#
#
#
# Constraints:
#
#
#       1 <= d, f <= 30
#       1 <= target <= 1000# Time:  O(d * f * t)
# Space: O(t)

class Solution(object):
    def numRollsToTarget(self, d, f, target):
```

```python
"""
:type d: int
:type f: int
:type target: int
:rtype: int
"""
MOD = 10**9+7
dp = [[0 for _ in xrange(target+1)] for _ in xrange(2)]
dp[0][0] = 1
for i in xrange(1, d+1):
    dp[i%2] = [0 for _ in xrange(target+1)]
    for k in xrange(1, f+1):
        for j in xrange(k, target+1):
            dp[i%2][j] = (dp[i%2][j] + dp[(i-1)%2][j-k]) % MOD
return dp[d%2][target] % MOD
```

knight-dialer.py

```
# The chess knight has a unique movement, it may move two squares vertically and
# one square horizontally, or two squares horizontally and one square vertically
# (with both forming the shape of an L). The possible movements of chess knight
# are shown in this diagaram:
#
# A chess knight can move as indicated in the chess diagram below:
#
# We have a chess knight and a phone pad as shown below, the knight can only
# stand on a numeric cell (i.e. blue cell).
#
# Given an integer n, return how many distinct phone numbers of length n we can
# dial.
#
# You are allowed to place the knight on any numeric cell initially and then you
# should perform n - 1 jumps to dial a number of length n. All jumps should be
# valid knight jumps.
#
# As the answer may be very large, return the answer modulo 109 + 7.
#
#
# Example 1:
#
# Input: n = 1
# Output: 10
# Explanation: We need to dial a number of length 1, so placing the knight over
# any numeric cell of the 10 cells is sufficient.
#
#
# Example 2:
#
# Input: n = 2
# Output: 20
# Explanation: All the valid number we can dial are [04, 06, 16, 18, 27, 29, 34,
# 38, 40, 43, 49, 60, 61, 67, 72, 76, 81, 83, 92, 94]
#
#
# Example 3:
#
# Input: n = 3
# Output: 46
#
#
# Example 4:
#
# Input: n = 4
# Output: 104
#
#
# Example 5:
#
# Input: n = 3131
# Output: 136006598
# Explanation: Please take care of the mod.
#
#
#
# Constraints:
#
```

```python
#
#        1 <= n <= 5000# Time:  O(logn)
# Space: O(1)

import itertools


class Solution(object):
    def knightDialer(self, N):
        """
        :type N: int
        :rtype: int
        """
        def matrix_expo(A, K):
            result = [[int(i==j) for j in xrange(len(A))] \
                      for i in xrange(len(A))]
            while K:
                if K % 2:
                    result = matrix_mult(result, A)
                A = matrix_mult(A, A)
                K /= 2
            return result

        def matrix_mult(A, B):
            ZB = zip(*B)
            return [[sum(a*b for a, b in itertools.izip(row, col)) % M \
                     for col in ZB] for row in A]

        M = 10**9 + 7
        T = [[0, 0, 0, 0, 1, 0, 1, 0, 0, 0],
             [0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
             [0, 0, 0, 0, 0, 0, 0, 1, 0, 1],
             [0, 0, 0, 0, 1, 0, 0, 0, 1, 0],
             [1, 0, 0, 1, 0, 0, 0, 0, 0, 1],
             [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
             [1, 1, 0, 0, 0, 0, 0, 1, 0, 0],
             [0, 0, 1, 0, 0, 0, 1, 0, 0, 0],
             [0, 1, 0, 1, 0, 0, 0, 0, 0, 0],
             [0, 0, 1, 0, 1, 0, 0, 0, 0, 0]]
        return sum(map(sum, matrix_expo(T, N-1))) % M


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def knightDialer(self, N):
        """
        :type N: int
        :rtype: int
        """
        M = 10**9 + 7
        moves = [[4, 6], [6, 8], [7, 9], [4, 8], [3, 9, 0], [],
                 [1, 7, 0], [2, 6], [1, 3], [2, 4]]

        dp = [[1 for _ in xrange(10)] for _ in xrange(2)]
        for i in xrange(N-1):
            dp[(i+1) % 2] = [0] * 10
            for j in xrange(10):
                for nei in moves[j]:
                    dp[(i+1) % 2][nei] += dp[i % 2][j]
```

```
            dp[(i+1) % 2][nei] %= M
    return sum(dp[(N-1) % 2]) % M
```

maximum-difference-between-node-and-ancestor.py

```python
# Given the root of a binary tree, find the maximum value V for which there
# exists different nodes A and B where V = |A.val - B.val| and A is an ancestor of
# B.
#
# (A node A is an ancestor of B if either: any child of A is equal to B, or any
# child of A is an ancestor of B.)
#
#
#
# Example 1:
#
#
#
# Input: [8,3,10,1,6,null,14,null,null,4,7,13]
# Output: 7
# Explanation:
# We have various ancestor-node differences, some of which are given below :
# |8 - 3| = 5
# |3 - 7| = 4
# |8 - 1| = 7
# |10 - 13| = 3
# Among all possible differences, the maximum value of 7 is obtained by |8 - 1|
# = 7.
#
#
#
#
# Note:
#
#
#       The number of nodes in the tree is between 2 and 5000.
#       Each node will have value between 0 and 100000.# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


# iterative stack solution
class Solution(object):
    def maxAncestorDiff(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        result = 0
        stack = [(root, 0, float("inf"))]
        while stack:
            node, mx, mn = stack.pop()
            if not node:
                continue
            result = max(result, mx-node.val, node.val-mn)
            mx = max(mx, node.val)
            mn = min(mn, node.val)
```

```python
            stack.append((node.left, mx, mn))
            stack.append((node.right, mx, mn))
        return result


# Time:  O(n)
# Space: O(h)
# recursive solution
class Solution2(object):
    def maxAncestorDiff(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def maxAncestorDiffHelper(node, mx, mn):
            if not node:
                return 0
            result = max(mx-node.val, node.val-mn)
            mx = max(mx, node.val)
            mn = min(mn, node.val)
            result = max(result, maxAncestorDiffHelper(node.left, mx, mn))
            result = max(result, maxAncestorDiffHelper(node.right, mx, mn))
            return result

        return maxAncestorDiffHelper(root, 0, float("inf"))
```

maximum-product-of-word-lengths.py

```python
# 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:
#
# Input: ["abcw","baz","foo","bar","xtfn","abcdef"]
# Output: 16
# Explanation: The two words can be "abcw", "xtfn".
#
# Example 2:
#
# Input: ["a","ab","abc","d","cd","bcd","abcd"]
# Output: 4
# Explanation: The two words can be "ab", "cd".
#
# Example 3:
#
# Input: ["a","aa","aaa","aaaa"]
# Output: 0
# Explanation: No such pair of words.
#
#
#
# Constraints:
#
#
#       0 <= words.length <= 10^3
#       0 <= words[i].length <= 10^3
#       words[i] consists only of lowercase English letters.# Time:  O(n) ~ O(n^2)
# Space: O(n)

class Solution(object):
    def maxProduct(self, words):
        """
        :type words: List[str]
        :rtype: int
        """
        def counting_sort(words):
            k = 1000  # k is max length of words in the dictionary
            buckets = [[] for _ in xrange(k)]
            for word in words:
                buckets[len(word)].append(word)
            res = []
            for i in reversed(xrange(k)):
                if buckets[i]:
                    res += buckets[i]
            return res

        words = counting_sort(words)
        bits = [0] * len(words)
        for i, word in enumerate(words):
            for c in word:
                bits[i] |= (1 << (ord(c) - ord('a')))

        max_product = 0
        for i in xrange(len(words) - 1):
```

```python
            if len(words[i]) ** 2 <= max_product:
                break
            for j in xrange(i + 1, len(words)):
                if len(words[i]) * len(words[j]) <= max_product:
                    break
                if not (bits[i] & bits[j]):
                    max_product = len(words[i]) * len(words[j])
        return max_product

# Time:  O(nlogn) ~ O(n^2)
# Space: O(n)
# Sorting + Pruning + Bit Manipulation
class Solution2(object):
    def maxProduct(self, words):
        """
        :type words: List[str]
        :rtype: int
        """
        words.sort(key=lambda x: len(x), reverse=True)
        bits = [0] * len(words)
        for i, word in enumerate(words):
            for c in word:
                bits[i] |= (1 << (ord(c) - ord('a')))

        max_product = 0
        for i in xrange(len(words) - 1):
            if len(words[i]) ** 2 <= max_product:
                break
            for j in xrange(i + 1, len(words)):
                if len(words[i]) * len(words[j]) <= max_product:
                    break
                if not (bits[i] & bits[j]):
                    max_product = len(words[i]) * len(words[j])
        return max_product
```

remove-duplicates-from-sorted-list-ii.py

```python
# Given a sorted linked list, delete all nodes that have duplicate numbers,
# leaving only distinct numbers from the original list.
#
# Return the linked list sorted as well.
#
# Example 1:
#
# Input: 1->2->3->3->4->4->5
# Output: 1->2->5
#
#
# Example 2:
#
# Input: 1->1->1->2->3
# Output: 2->3# Time:  O(n)
# Space: O(1)

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

    def __repr__(self):
        if self is None:
            return "Nil"
        else:
            return "{} -> {}".format(self.val, repr(self.next))

class Solution(object):
    def deleteDuplicates(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        dummy = ListNode(0)
        pre, cur = dummy, head
        while cur:
            if cur.next and cur.next.val == cur.val:
                val = cur.val
                while cur and cur.val == val:
                    cur = cur.next
                pre.next = cur
            else:
                pre.next = cur
                pre = cur
                cur = cur.next
        return dummy.next
```

longest-arithmetic-subsequence-of-given-difference.py

```
# Given an integer array arr and an integer difference, return the length of the
# longest subsequence in arr which is an arithmetic sequence such that the
# difference between adjacent elements in the subsequence equals difference.
#
#
# Example 1:
#
# Input: arr = [1,2,3,4], difference = 1
# Output: 4
# Explanation: The longest arithmetic subsequence is [1,2,3,4].
#
# Example 2:
#
# Input: arr = [1,3,5,7], difference = 1
# Output: 1
# Explanation: The longest arithmetic subsequence is any single element.
#
#
# Example 3:
#
# Input: arr = [1,5,7,8,5,3,4,2,1], difference = -2
# Output: 4
# Explanation: The longest arithmetic subsequence is [7,5,3,1].
#
#
#
# Constraints:
#
#
#       1 <= arr.length <= 10^5
#       -10^4 <= arr[i], difference <= 10^4# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def longestSubsequence(self, arr, difference):
        """
        :type arr: List[int]
        :type difference: int
        :rtype: int
        """
        result = 1
        lookup = collections.defaultdict(int)
        for i in xrange(len(arr)):
            lookup[arr[i]] = lookup[arr[i]-difference] + 1
            result = max(result, lookup[arr[i]])
        return result
```

sum-of-two-integers.py

```python
# Example 2:
#
# Input: a = -2, b = 3
# Output: 1# Time:  O(1)
# Space: O(1)

class Solution(object):
    def getSum(self, a, b):
        """
        :type a: int
        :type b: int
        :rtype: int
        """
        bit_length = 32
        neg_bit, mask = (1 << bit_length) >> 1, ~(~0 << bit_length)

        a = (a | ~mask) if (a & neg_bit) else (a & mask)
        b = (b | ~mask) if (b & neg_bit) else (b & mask)

        while b:
            carry = a & b
            a ^= b
            a = (a | ~mask) if (a & neg_bit) else (a & mask)
            b = carry << 1
            b = (b | ~mask) if (b & neg_bit) else (b & mask)

        return a

    def getSum2(self, a, b):
        """
        :type a: int
        :type b: int
        :rtype: int
        """
        # 32 bits integer max
        MAX = 0x7FFFFFFF
        # 32 bits interger min
        MIN = 0x80000000
        # mask to get last 32 bits
        mask = 0xFFFFFFFF
        while b:
            # ^ get different bits and & gets double 1s, << moves carry
            a, b = (a ^ b) & mask, ((a & b) << 1) & mask
        # if a is negative, get a's 32 bits complement positive first
        # then get 32-bit positive's Python complement negative
        return a if a <= MAX else ~(a ^ mask)

    def minus(self, a, b):
        b = self.getSum(~b, 1)
        return self.getSum(a, b)

    def multiply(self, a, b):
        isNeg = (a > 0) ^ (b > 0)
        x = a if a > 0 else self.getSum(~a, 1)
        y = b if b > 0 else self.getSum(~b, 1)
        ans = 0
        while y & 0x01:
            ans = self.getSum(ans, x)
```

```python
            y >>= 1
            x <<= 1
        return self.getSum(~ans, 1) if isNeg else ans

    def divide(self, a, b):
        isNeg = (a > 0) ^ (b > 0)
        x = a if a > 0 else self.getSum(~a, 1)
        y = b if b > 0 else self.getSum(~b, 1)
        ans = 0
        for i in range(31, -1, -1):
            if (x >> i) >= y:
                x = self.minus(x, y << i)
                ans = self.getSum(ans, 1 << i)
        return self.getSum(~ans, 1) if isNeg else ans
```

partition-equal-subset-sum.py

```
# Given a non-empty array containing only positive integers, find if the array
# can be partitioned into two subsets such that the sum of elements in both
# subsets is equal.
#
# Note:
#
#
#       Each of the array element will not exceed 100.
#       The array size will not exceed 200.
#
#
#
#
# Example 1:
#
# Input: [1, 5, 11, 5]
#
# Output: true
#
# Explanation: The array can be partitioned as [1, 5, 5] and [11].
#
#
#
#
# Example 2:
#
# Input: [1, 2, 3, 5]
#
# Output: false
#
# Explanation: The array cannot be partitioned into equal sum subsets.# Time:  O(n * s), s is the sum of nums
# Space: O(s)

class Solution(object):
    def canPartition(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        s = sum(nums)
        if s % 2:
            return False

        dp = [False] * (s/2 + 1)
        dp[0] = True
        for num in nums:
            for i in reversed(xrange(1, len(dp))):
                if num <= i:
                    dp[i] = dp[i] or dp[i - num]
        return dp[-1]
```

moving-stones-until-consecutive-ii.py

```
# Example 2:
#
# Input: [6,5,4,3,10]
# Output: [2,3]
# We can move 3 -> 8 then 10 -> 7 to finish the game.
# Or, we can move 3 -> 7, 4 -> 8, 5 -> 9 to finish the game.
# Notice we cannot move 10 -> 2 to finish the game, because that would be an
# illegal move.
#
#
#
# Example 3:
#
# Input: [100,101,104,102,103]
# Output: [0,0]# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def numMovesStonesII(self, stones):
        """
        :type stones: List[int]
        :rtype: List[int]
        """
        stones.sort()
        left, min_moves = 0, float("inf")
        max_moves = max(stones[-1]-stones[1], stones[-2]-stones[0]) - (len(stones)-2)
        for right in xrange(len(stones)):
            while stones[right]-stones[left]+1 > len(stones): # find window size <= len(stones)
                left += 1
            if len(stones)-(right-left+1) == 1 and stones[right]-stones[left]+1 == len(stones)-1:
                min_moves = min(min_moves, 2)  # case (1, 2, 3, 4), 7
            else:
                min_moves = min(min_moves, len(stones)-(right-left+1))  # move stones not in this window
        return [min_moves, max_moves]
```

corporate-flight-bookings.py

```
# There are n flights, and they are labeled from 1 to n.
#
# We have a list of flight bookings.  The i-th booking bookings[i] = [i, j,
# k] means that we booked k seats from flights labeled i to j inclusive.
#
# Return an array answer of length n, representing the number of seats booked on
# each flight in order of their label.
#
#
# Example 1:
#
# Input: bookings = [[1,2,10],[2,3,20],[2,5,25]], n = 5
# Output: [10,55,45,25,25]
#
#
#
# Constraints:
#
#
#       1 <= bookings.length <= 20000
#       1 <= bookings[i][0] <= bookings[i][1] <= n <= 20000
#       1 <= bookings[i][2] <= 10000# Time:  O(n)
# Space: O(1)

class Solution(object):
    def corpFlightBookings(self, bookings, n):
        """
        :type bookings: List[List[int]]
        :type n: int
        :rtype: List[int]
        """
        result = [0]*(n+1)
        for i, j, k in bookings:
            result[i-1] += k
            result[j] -= k
        for i in xrange(1, len(result)):
            result[i] += result[i-1]
        result.pop()
        return result
```

print-binary-tree.py

```python
# Print a binary tree in an m*n 2D string array following these rules:
#
#
# The row number m should be equal to the height of the given binary tree.
# The column number n should always be an odd number.
# The root node's value (in string format) should be put in the exactly middle
# of the first row it can be put. The column and the row where the root node
# belongs will separate the rest space into two parts (left-bottom part and right-
# bottom part). You should print the left subtree in the left-bottom part and
# print the right subtree in the right-bottom part. The left-bottom part and the
# right-bottom part should have the same size. Even if one subtree is none while
# the other is not, you don't need to print anything for the none subtree but
# still need to leave the space as large as that for the other subtree. However,
# if two subtrees are none, then you don't need to leave space for both of them.
# Each unused space should contain an empty string "".
# Print the subtrees following the same rules.
#
#
# Example 1:
#
# Input:
#      1
#     /
#    2
# Output:
# [["", "1", ""],
#  ["2", "", ""]]
#
#
#
#
# Example 2:
#
# Input:
#      1
#     / \
#    2   3
#     \
#      4
# Output:
# [["", "", "", "1", "", "", ""],
#  ["", "2", "", "", "", "3", ""],
#  ["", "", "4", "", "", "", ""]]
#
#
#
# Example 3:
#
# Input:
#       1
#      / \
#     2   5
#    /
#   3
#  /
# 4
# Output:
#
```

```python
# [["",  "",  "", "",  "", "", "", "1", "",  "",  "",  "",  "", "", ""]
#  ["",  "",  "", "2", "", "", "", "",  "",  "",  "",  "5", "", "", ""]
#  ["",  "3", "", "",  "", "", "", "",  "",  "",  "",  "",  "", "", ""]
#  ["4", "",  "", "",  "", "", "", "",  "",  "",  "",  "",  "", "", ""]]
#
#
#
# Note:
# The height of binary tree is in the range of [1, 10].# Time:  O(h * 2^h)
# Space: O(h * 2^h)

class Solution(object):
    def printTree(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[str]]
        """
        def getWidth(root):
            if not root:
                return 0
            return 2 * max(getWidth(root.left), getWidth(root.right)) + 1

        def getHeight(root):
            if not root:
                return 0
            return max(getHeight(root.left), getHeight(root.right)) + 1

        def preorderTraversal(root, level, left, right, result):
            if not root:
                return
            mid = left + (right-left)/2
            result[level][mid] = str(root.val)
            preorderTraversal(root.left, level+1, left, mid-1, result)
            preorderTraversal(root.right, level+1, mid+1, right, result)

        h, w = getHeight(root), getWidth(root)
        result = [[""] * w for _ in xrange(h)]
        preorderTraversal(root, 0, 0, w-1, result)
        return result
```

linked-list-random-node.py

```python
# Given a singly linked list, return a random node's value from the linked list.
# Each node must have the same probability of being chosen.
#
# Follow up:
#
# What if the linked list is extremely large and its length is unknown to you?
# Could you solve this efficiently without using extra space?
#
#
# Example:
# // Init a singly linked list [1,2,3].
# ListNode head = new ListNode(1);
# head.next = new ListNode(2);
# head.next.next = new ListNode(3);
# Solution solution = new Solution(head);
#
# // getRandom() should return either 1, 2, or 3 randomly. Each element should
# have equal probability of returning.
# solution.getRandom();# Time:  O(n)
# Space: O(1)

from random import randint

class Solution(object):

    def __init__(self, head):
        """
        @param head The linked list's head. Note that the head is guanranteed to be not null, so it contains a
        :type head: ListNode
        """
        self.__head = head


    # Proof of Reservoir Sampling:
    # https://discuss.leetcode.com/topic/53753/brief-explanation-for-reservoir-sampling
    def getRandom(self):
        """
        Returns a random node's value.
        :rtype: int
        """
        reservoir = -1
        curr, n = self.__head, 0
        while curr:
            reservoir = curr.val if randint(1, n+1) == 1 else reservoir
            curr, n = curr.next, n+1
        return reservoir
```

swap-for-longest-repeated-character-substring.py

```python
# Given a string text, we are allowed to swap two of the characters in the
# string. Find the length of the longest substring with repeated characters.
#
#
# Example 1:
#
# Input: text = "ababa"
# Output: 3
# Explanation: We can swap the first 'b' with the last 'a', or the last 'b' with
# the first 'a'. Then, the longest repeated character substring is "aaa", which
# its length is 3.
#
#
# Example 2:
#
# Input: text = "aaabaaa"
# Output: 6
# Explanation: Swap 'b' with the last 'a' (or the first 'a'), and we get longest
# repeated character substring "aaaaaa", which its length is 6.
#
#
# Example 3:
#
# Input: text = "aaabbaaa"
# Output: 4
#
#
# Example 4:
#
# Input: text = "aaaaa"
# Output: 5
# Explanation: No need to swap, longest repeated character substring is "aaaaa",
# length is 5.
#
#
# Example 5:
#
# Input: text = "abcdef"
# Output: 1
#
#
#
# Constraints:
#
#
#       1 <= text.length <= 20000
#       text consist of lowercase English characters only.# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def maxRepOpt1(self, text):
        """
        :type text: str
        :rtype: int
        """
```

```python
        K = 1
        result = 0
        total_count, count = collections.Counter(), collections.Counter()
        left, max_count = 0, 0
        for i in xrange(len(text)):
            total_count[text[i]] += 1
            count[text[i]] += 1
            max_count = max(max_count, count[text[i]])
            if i-left+1 - max_count > K:
                count[text[left]] -= 1
                left += 1
            result = max(result, min(i-left+1, total_count[text[i]]))
        return result



# Time:  O(n)
# Space: O(n)
import itertools


class Solution2(object):
    def maxRepOpt1(self, text):
        """
        :type text: str
        :rtype: int
        """
        A = [[c, len(list(group))] for c, group in itertools.groupby(text)]
        total_count = collections.Counter(text)
        result = max(min(l+1, total_count[c]) for c, l in A)
        for i in xrange(1, len(A)-1):
            if A[i-1][0] == A[i+1][0] and A[i][1] == 1:
                result = max(result, min(A[i-1][1] + 1 + A[i+1][1], total_count[A[i+1][0]]))
        return result
```

maximum-sum-of-two-non-overlapping-subarrays.py

```
# Example 2:
#
# Input: A = [3,8,1,3,2,1,8,9,0], L = 3, M = 2
# Output: 29
# Explanation: One choice of subarrays is [3,8,1] with length 3, and [8,9] with
# length 2.
#
#
#
# Example 3:
#
# Input: A = [2,1,5,6,0,9,5,0,3,8], L = 4, M = 3
# Output: 31
# Explanation: One choice of subarrays is [5,6,0,9] with length 4, and [3,8]
# with length 3.
#
#
#
#
# Note:
#
#
#       L >= 1
#       M >= 1
#       L + M <= A.length <= 1000
#       0 <= A[i] <= 1000# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxSumTwoNoOverlap(self, A, L, M):
        """
        :type A: List[int]
        :type L: int
        :type M: int
        :rtype: int
        """
        for i in xrange(1, len(A)):
            A[i] += A[i-1]
        result, L_max, M_max = A[L+M-1], A[L-1], A[M-1]
        for i in xrange(L+M, len(A)):
            L_max = max(L_max, A[i-M] - A[i-L-M])
            M_max = max(M_max, A[i-L] - A[i-L-M])
            result = max(result,
                         L_max + A[i] - A[i-M],
                         M_max + A[i] - A[i-L])
        return result
```

binary-tree-level-order-traversal.py

```
# 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],
#
#     3
#    / \
#   9  20
#     /  \
#    15   7
#
#
#
# return its level order traversal as:
#
# [
#   [3],
#   [9,20],
#   [15,7]
# ]# Time:  O(n)
# Space: O(n)

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    # @param root, a tree node
    # @return a list of lists of integers
    def levelOrder(self, root):
        if root is None:
            return []
        result, current = [], [root]
        while current:
            next_level, vals = [], []
            for node in current:
                vals.append(node.val)
                if node.left:
                    next_level.append(node.left)
                if node.right:
                    next_level.append(node.right)
            current = next_level
            result.append(vals)
        return result
```

construct-binary-search-tree-from-preorder-traversal.py

```python
# Return the root node of a binary search tree that matches the given preorder
# traversal.
#
# (Recall that a binary search tree is a binary tree where for every node, any
# descendant of node.left has a value < node.val, and any descendant of node.right
# has a value > node.val.  Also recall that a preorder traversal displays the
# value of the node first, then traverses node.left, then traverses node.right.)
#
# It's guaranteed that for the given test cases there is always possible to find
# a binary search tree with the given requirements.
#
# Example 1:
#
# Input: [8,5,1,7,10,12]
# Output: [8,5,10,1,7,null,12]
#
#
#
#
# Constraints:
#
#
#       1 <= preorder.length <= 100
#       1 <= preorder[i] <= 10^8
#       The values of preorder are distinct.# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def bstFromPreorder(self, preorder):
        """
        :type preorder: List[int]
        :rtype: TreeNode
        """
        def bstFromPreorderHelper(preorder, left, right, index):
            if index[0] == len(preorder) or \
               preorder[index[0]] < left or \
               preorder[index[0]] > right:
                return None

            root = TreeNode(preorder[index[0]])
            index[0] += 1
            root.left = bstFromPreorderHelper(preorder, left, root.val, index)
            root.right = bstFromPreorderHelper(preorder, root.val, right, index)
            return root

        return bstFromPreorderHelper(preorder, float("-inf"), float("inf"), [0])
```

minimum-number-of-arrows-to-burst-balloons.py

```
# There are a number of spherical balloons spread in two-dimensional space. For
# each balloon, provided input is the start and end coordinates of the horizontal
# diameter. Since it's horizontal, y-coordinates don't matter and hence the
# x-coordinates of start and end of the diameter suffice. Start is always smaller
# than end. There will be at most 104 balloons.
#
# An arrow can be shot up exactly vertically from different points along the
# x-axis. A balloon with xstart and xend bursts by an arrow shot at x if xstart
# x  xend. There is no limit to the number of arrows that can be shot. An arrow
# once shot keeps travelling up infinitely. The problem is to find the minimum
# number of arrows that must be shot to burst all balloons.
#
# Example:
#
# Input:
# [[10,16], [2,8], [1,6], [7,12]]
#
# Output:
# 2
#
# Explanation:
# One way is to shoot one arrow for example at x = 6 (bursting the balloons
# [2,8] and [1,6]) and another arrow at x = 11 (bursting the other two balloons).# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def findMinArrowShots(self, points):
        """
        :type points: List[List[int]]
        :rtype: int
        """
        if not points:
            return 0

        points.sort()

        result = 0
        i = 0
        while i < len(points):
            j = i + 1
            right_bound = points[i][1]
            while j < len(points) and points[j][0] <= right_bound:
                right_bound = min(right_bound, points[j][1])
                j += 1
            result += 1
            i = j
        return result
```

search-a-2d-matrix-ii.py

```python
# 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.
#
#
# Example:
#
# Consider the following matrix:
#
# [
#   [1,   4,  7, 11, 15],
#   [2,   5,  8, 12, 19],
#   [3,   6,  9, 16, 22],
#   [10, 13, 14, 17, 24],
#   [18, 21, 23, 26, 30]
# ]
#
#
# Given target = 5, return true.
#
# Given target = 20, return false.# Time:  O(m + n)
# Space: O(1)

class Solution(object):
    # @param {integer[][]} matrix
    # @param {integer} target
    # @return {boolean}
    def searchMatrix(self, matrix, target):
        m = len(matrix)
        if m == 0:
            return False

        n = len(matrix[0])
        if n == 0:
            return False

        i, j = 0, n - 1
        while i < m and j >= 0:
            if matrix[i][j] == target:
                return True
            elif matrix[i][j] > target:
                j -= 1
            else:
                i += 1

        return False
```

mini-parser.py

```
# Given a nested list of integers represented as a string, implement a parser to
# deserialize it.
#
# Each element is either an integer, or a list -- whose elements may also be
# integers or other lists.
#
# Note: You may assume that the string is well-formed:
#
#
#       String is non-empty.
#       String does not contain white spaces.
#       String contains only digits 0-9, [, - ,, ].
#
#
#
#
# Example 1:
#
# Given s = "324",
#
# You should return a NestedInteger object which contains a single integer 324.
#
#
#
#
# Example 2:
#
# Given s = "[123,[456,[789]]]",
#
# Return a NestedInteger object containing a nested list with 2 elements:
#
# 1. An integer containing value 123.
# 2. A nested list containing two elements:
#     i.  An integer containing value 456.
#     ii. A nested list with one element:
#          a. An integer containing value 789.# Time:  O(n)
# Space: O(h)

class NestedInteger(object):
   def __init__(self, value=None):
       """
       If value is not specified, initializes an empty list.
       Otherwise initializes a single integer equal to value.
       """

   def isInteger(self):
       """
       @return True if this NestedInteger holds a single integer, rather than a nested list.
       :rtype bool
       """

   def add(self, elem):
       """
       Set this NestedInteger to hold a nested list and adds a nested integer elem to it.
       :rtype void
       """

   def setInteger(self, value):
```

```python
        """
        Set this NestedInteger to hold a single integer equal to value.
        :rtype void
        """

    def getInteger(self):
        """
        @return the single integer that this NestedInteger holds, if it holds a single integer
        Return None if this NestedInteger holds a nested list
        :rtype int
        """

    def getList(self):
        """
        @return the nested list that this NestedInteger holds, if it holds a nested list
        Return None if this NestedInteger holds a single integer
        :rtype List[NestedInteger]
        """


class Solution(object):
    def deserialize(self, s):
        if not s:
            return NestedInteger()

        if s[0] != '[':
            return NestedInteger(int(s))

        stk = []

        i = 0
        for j in xrange(len(s)):
            if s[j] == '[':
                stk += NestedInteger(),
                i = j+1
            elif s[j] in ',]':
                if s[j-1].isdigit():
                    stk[-1].add(NestedInteger(int(s[i:j])))
                if s[j] == ']' and len(stk) > 1:
                    cur = stk[-1]
                    stk.pop()
                    stk[-1].add(cur)
                i = j+1

        return stk[-1]
```

angle-between-hands-of-a-clock.py

```
# Given two numbers, hour and minutes. Return the smaller angle (in degrees)
# formed between the hour and the minute hand.
#
#
# Example 1:
#
#
#
# Input: hour = 12, minutes = 30
# Output: 165
#
#
# Example 2:
#
#
#
# Input: hour = 3, minutes = 30
# Output: 75
#
#
# Example 3:
#
#
#
# Input: hour = 3, minutes = 15
# Output: 7.5
#
#
# Example 4:
#
# Input: hour = 4, minutes = 50
# Output: 155
#
#
# Example 5:
#
# Input: hour = 12, minutes = 0
# Output: 0
#
#
#
# Constraints:
#
#
#       1 <= hour <= 12
#       0 <= minutes <= 59
#       Answers within 10^-5 of the actual value will be accepted as correct.# Time:  O(1)
# Space: O(1)

class Solution(object):
    def angleClock(self, hour, minutes):
        """
        :type hour: int
        :type minutes: int
        :rtype: float
        """
        angle1 = (hour % 12 * 60.0 + minutes) / 720.0
        angle2 = minutes / 60.0
```

```
diff = abs(angle1-angle2)
return min(diff, 1.0-diff) * 360.0
```

out-of-boundary-paths.py

```python
# There is an m by n grid with a ball. Given the start coordinate (i,j) of the
# ball, you can move the ball to adjacent cell or cross the grid boundary in four
# directions (up, down, left, right). However, you can at most move N times. Find
# out the number of paths to move the ball out of grid boundary. The answer may be
# very large, return it after mod 109 + 7.
#
#
#
# Example 1:
#
# Input: m = 2, n = 2, N = 2, i = 0, j = 0
# Output: 6
# Explanation:
#
#
#
# Example 2:
#
# Input: m = 1, n = 3, N = 3, i = 0, j = 1
# Output: 12
# Explanation:
#
#
#
#
#
# Note:
#
#
#       Once you move the ball out of boundary, you cannot move it back.
#       The length and height of the grid is in range [1,50].
#       N is in range [0,50].# Time:  O(N * m * n)
# Space: O(m * n)

class Solution(object):
    def findPaths(self, m, n, N, x, y):
        """
        :type m: int
        :type n: int
        :type N: int
        :type x: int
        :type y: int
        :rtype: int
        """
        M = 1000000000 + 7
        dp = [[[0 for _ in xrange(n)] for _ in xrange(m)] for _ in xrange(2)]
        for moves in xrange(N):
            for i in xrange(m):
                for j in xrange(n):
                    dp[(moves + 1) % 2][i][j] = (((1 if (i == 0) else dp[moves % 2][i - 1][j]) + \
                                                  (1 if (i == m - 1) else dp[moves % 2][i + 1][j])) % M + \
                                                 ((1 if (j == 0) else dp[moves % 2][i][j - 1]) + \
                                                  (1 if (j == n - 1) else dp[moves % 2][i][j + 1])) % M) % M
        return dp[N % 2][x][y]
```

battleships-in-a-board.py

```python
# Given an 2D board, count how many battleships are in it. The battleships are
# represented with 'X's, empty slots are represented with '.'s. You may assume the
# following rules:
#
#
# You receive a valid board, made of only battleships or empty slots.
# Battleships can only be placed horizontally or vertically. In other words,
# they can only be made of the shape 1xN (1 row, N columns) or Nx1 (N rows, 1
# column), where N can be of any size.
# At least one horizontal or vertical cell separates between two battleships -
# there are no adjacent battleships.
#
#
# Example:
#
# X..X
# ...X
# ...X
#
# In the above board there are 2 battleships.
#
# Invalid Example:
#
# ...X
# XXXX
# ...X
#
# This is an invalid board that you will not receive - as battleships will
# always have a cell separating between them.
#
# Follow up:
# Could you do it in one-pass, using only O(1) extra memory and without
# modifying the value of the board?# Time:  O(m * n)
# Space: O(1)


class Solution(object):
    def countBattleships(self, board):
        """
        :type board: List[List[str]]
        :rtype: int
        """
        if not board or not board[0]:
            return 0

        cnt = 0
        for i in xrange(len(board)):
            for j in xrange(len(board[0])):
                cnt += int(board[i][j] == 'X' and
                           (i == 0 or board[i - 1][j] != 'X') and
                           (j == 0 or board[i][j - 1] != 'X'))
        return cnt
```

subarray-sum-equals-k.py

```python
# Given an array of integers and an integer k, you need to find the total number
# of continuous subarrays whose sum equals to k.
#
# Example 1:
#
# Input:nums = [1,1,1], k = 2
# Output: 2
#
#
#
# Constraints:
#
#
#       The length of the array is in range [1, 20,000].
#       The range of numbers in the array is [-1000, 1000] and the range of the
# integer k is [-1e7, 1e7].# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def subarraySum(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: int
        """
        result = 0
        accumulated_sum = 0
        lookup = collections.defaultdict(int)
        lookup[0] += 1
        for num in nums:
            accumulated_sum += num
            result += lookup[accumulated_sum - k]
            lookup[accumulated_sum] += 1
        return result
```

set-matrix-zeroes.py

```
# Given an m x n matrix. If an element is 0, set its entire row and column to 0.
# Do it in-place.
#
# Follow up:
#
#
#       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?
#
#
#
# Example 1:
#
# Input: matrix = [[1,1,1],[1,0,1],[1,1,1]]
# Output: [[1,0,1],[0,0,0],[1,0,1]]
#
#
# Example 2:
#
# Input: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
# Output: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]
#
#
#
# Constraints:
#
#
#       m == matrix.length
#       n == matrix[0].length
#       1 <= m, n <= 200
#       -10^9 <= matrix[i][j] <= 10^9from functools import reduce
# Time:  O(m * n)
# Space: O(1)

class Solution(object):
    # @param matrix, a list of lists of integers
    # RETURN NOTHING, MODIFY matrix IN PLACE.
    def setZeroes(self, matrix):
        first_col = reduce(lambda acc, i: acc or matrix[i][0] == 0, xrange(len(matrix)), False)
        first_row = reduce(lambda acc, j: acc or matrix[0][j] == 0, xrange(len(matrix[0])), False)

        for i in xrange(1, len(matrix)):
            for j in xrange(1, len(matrix[0])):
                if matrix[i][j] == 0:
                    matrix[i][0], matrix[0][j] = 0, 0

        for i in xrange(1, len(matrix)):
            for j in xrange(1, len(matrix[0])):
                if matrix[i][0] == 0 or matrix[0][j] == 0:
                    matrix[i][j] = 0

        if first_col:
            for i in xrange(len(matrix)):
                matrix[i][0] = 0

        if first_row:
```

```
for j in xrange(len(matrix[0])):
    matrix[0][j] = 0
```

best-sightseeing-pair.py

```python
# Given an array A of positive integers, A[i] represents the value of the i-th
# sightseeing spot, and two sightseeing spots i and j have distance j - i between
# them.
#
# The score of a pair (i < j) of sightseeing spots is (A[i] + A[j] + i - j) :
# the sum of the values of the sightseeing spots, minus the distance between them.
#
# Return the maximum score of a pair of sightseeing spots.
#
#
#
# Example 1:
#
# Input: [8,1,5,2,6]
# Output: 11
# Explanation: i = 0, j = 2, A[i] + A[j] + i - j = 8 + 5 + 0 - 2 = 11
#
#
#
#
# Note:
#
#
#       2 <= A.length <= 50000
#       1 <= A[i] <= 1000# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxScoreSightseeingPair(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        result, curr = 0, 0
        for x in A:
            result = max(result, curr+x)
            curr = max(curr, x)-1
        return result
```

find-k-pairs-with-smallest-sums.py

```python
# You are given two integer arrays nums1 and nums2 sorted in ascending order and
# an integer k.
#
# Define a pair (u,v) which consists of one element from the first array and one
# element from the second array.
#
# Find the k pairs (u1,v1),(u2,v2) ...(uk,vk) with the smallest sums.
#
# Example 1:
#
# Input: nums1 = [1,7,11], nums2 = [2,4,6], k = 3
# Output: [[1,2],[1,4],[1,6]]
# Explanation: The first 3 pairs are returned from the sequence:
#              [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]
#
# Example 2:
#
# Input: nums1 = [1,1,2], nums2 = [1,2,3], k = 2
# Output: [1,1],[1,1]
# Explanation: The first 2 pairs are returned from the sequence:
#              [1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]
#
# Example 3:
#
# Input: nums1 = [1,2], nums2 = [3], k = 3
# Output: [1,3],[2,3]
# Explanation: All possible pairs are returned from the sequence: [1,3],[2,3]# Time:  O(k * log(min(n, m, k)))
# Space: O(min(n, m, k))

from heapq import heappush, heappop

class Solution(object):
    def kSmallestPairs(self, nums1, nums2, k):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :type k: int
        :rtype: List[List[int]]
        """
        pairs = []
        if len(nums1) > len(nums2):
            tmp = self.kSmallestPairs(nums2, nums1, k)
            for pair in tmp:
                pairs.append([pair[1], pair[0]])
            return pairs

        min_heap = []
        def push(i, j):
            if i < len(nums1) and j < len(nums2):
                heappush(min_heap, [nums1[i] + nums2[j], i, j])

        push(0, 0)
        while min_heap and len(pairs) < k:
            _, i, j = heappop(min_heap)
            pairs.append([nums1[i], nums2[j]])
            push(i, j + 1)
            if j == 0:
                push(i + 1, 0)  # at most queue min(n, m) space
```

```python
        return pairs


# time:  O(mn * log k)
# space: O(k)
from heapq import nsmallest
from itertools import product


class Solution2(object):
    def kSmallestPairs(self, nums1, nums2, k):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :type k: int
        :rtype: List[List[int]]
        """
        return nsmallest(k, product(nums1, nums2), key=sum)
```

distribute-coins-in-binary-tree.py

```
# Given the root of a binary tree with N nodes, each node in the tree has
# node.val coins, and there are N coins total.
#
# In one move, we may choose two adjacent nodes and move one coin from one node
# to another.  (The move may be from parent to child, or from child to parent.)
#
# Return the number of moves required to make every node have exactly one coin.
#
#
#
#
# Example 1:
#
#
#
# Input: [3,0,0]
# Output: 2
# Explanation: From the root of the tree, we move one coin to its left child,
# and one coin to its right child.
#
#
#
# Example 2:
#
#
#
# Input: [0,3,0]
# Output: 3
# Explanation: From the left child of the root, we move two coins to the root
# [taking two moves].  Then, we move one coin from the root of the tree to the
# right child.
#
#
#
# Example 3:
#
#
#
# Input: [1,0,2]
# Output: 2
#
#
#
# Example 4:
#
#
#
# Input: [1,0,0,null,3]
# Output: 4
#
#
#
#
# Note:
#
#
#       1<= N <= 100
#       0 <= node.val <= N# Time:  O(n)
```

```python
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def distributeCoins(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def dfs(root, result):
            if not root:
                return 0
            left, right = dfs(root.left, result), dfs(root.right, result)
            result[0] += abs(left) + abs(right)
            return root.val + left + right - 1

        result = [0]
        dfs(root, result)
        return result[0]
```

sort-characters-by-frequency.py

```python
# Given a string, sort it in decreasing order based on the frequency of
# characters.
#
# Example 1:
# Input:
# "tree"
#
# Output:
# "eert"
#
# Explanation:
# 'e' appears twice while 'r' and 't' both appear once.
# So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid
# answer.
#
#
#
# Example 2:
# Input:
# "cccaaa"
#
# Output:
# "cccaaa"
#
# Explanation:
# Both 'c' and 'a' appear three times, so "aaaccc" is also a valid answer.
# Note that "cacaca" is incorrect, as the same characters must be together.
#
#
#
# Example 3:
# Input:
# "Aabb"
#
# Output:
# "bbAa"
#
# Explanation:
# "bbaA" is also a valid answer, but "Aabb" is incorrect.
# Note that 'A' and 'a' are treated as two different characters.# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def frequencySort(self, s):
        """
        :type s: str
        :rtype: str
        """
        freq = collections.defaultdict(int)
        for c in s:
            freq[c] += 1

        counts = [""] * (len(s)+1)
        for c in freq:
            counts[freq[c]] += c
```

```
result = ""
for count in reversed(xrange(len(counts)-1)):
    for c in counts[count]:
        result += c * count

return result
```

minimum-area-rectangle.py

```
# Given a set of points in the xy-plane, determine the minimum area of a
# rectangle formed from these points, with sides parallel to the x and y axes.
#
# If there isn't any rectangle, return 0.
#
#
#
#
# Example 1:
#
# Input: [[1,1],[1,3],[3,1],[3,3],[2,2]]
# Output: 4
#
#
#
# Example 2:
#
# Input: [[1,1],[1,3],[3,1],[3,3],[4,1],[4,3]]
# Output: 2
#
#
#
#
# Note:
#
#
#       1 <= points.length <= 500
#       0 <= points[i][0] <= 40000
#       0 <= points[i][1] <= 40000
#       All points are distinct.# Time:  O(n^1.5) on average
#        O(n^2) on worst
# Space: O(n)

import collections


class Solution(object):
    def minAreaRect(self, points):
        """
        :type points: List[List[int]]
        :rtype: int
        """
        nx = len(set(x for x, y in points))
        ny = len(set(y for x, y in points))

        p = collections.defaultdict(list)
        if nx > ny:
            for x, y in points:
                p[x].append(y)
        else:
            for x, y in points:
                p[y].append(x)

        lookup = {}
        result = float("inf")
        for x in sorted(p):
            p[x].sort()
            for j in xrange(len(p[x])):
```

```python
                for i in xrange(j):
                    y1, y2 = p[x][i], p[x][j]
                    if (y1, y2) in lookup:
                        result = min(result, (x-lookup[y1, y2]) * abs(y2-y1))
                    lookup[y1, y2] = x
        return result if result != float("inf") else 0


# Time:  O(n^2)
# Space: O(n)
class Solution2(object):
    def minAreaRect(self, points):
        """
        :type points: List[List[int]]
        :rtype: int
        """
        lookup = set()
        result = float("inf")
        for x1, y1 in points:
            for x2, y2 in lookup:
                if (x1, y2) in lookup and (x2, y1) in lookup:
                    result = min(result, abs(x1-x2) * abs(y1-y2))
            lookup.add((x1, y1))
        return result if result != float("inf") else 0
```

sequential-digits.py

```python
# An integer has sequential digits if and only if each digit in the number is
# one more than the previous digit.
#
# Return a sorted list of all the integers in the range [low, high] inclusive
# that have sequential digits.
#
#
# Example 1:
# Input: low = 100, high = 300
# Output: [123,234]
# Example 2:
# Input: low = 1000, high = 13000
# Output: [1234,2345,3456,4567,5678,6789,12345]
#
#
# Constraints:
#
#
#       10 <= low <= high <= 10^9# Time:  O((8 + 1) * 8 / 2) = O(1)
# Space: O(8) = O(1)

import collections


class Solution(object):
    def sequentialDigits(self, low, high):
        """
        :type low: int
        :type high: int
        :rtype: List[int]
        """
        result = []
        q = collections.deque(range(1, 9))
        while q:
            num = q.popleft()
            if num > high:
                continue
            if low <= num:
                result.append(num)
            if num%10+1 < 10:
                q.append(num*10+num%10+1)
        return result
```

perfect-squares.py

```python
# Given a positive integer n, find the least number of perfect square numbers
# (for example, 1, 4, 9, 16, ...) which sum to n.
#
# Example 1:
#
# Input: n = 12
# Output: 3
# Explanation: 12 = 4 + 4 + 4.
#
# Example 2:
#
# Input: n = 13
# Output: 2
# Explanation: 13 = 4 + 9.# Time:  O(n * sqrt(n))
# Space: O(n)

class Solution(object):
    _num = [0]
    def numSquares(self, n):
        """
        :type n: int
        :rtype: int
        """
        num = self._num
        while len(num) <= n:
            num += min(num[-i*i] for i in xrange(1, int(len(num)**0.5+1))) + 1,
        return num[n]
```

longest-turbulent-subarray.py

```python
# Example 3:
#
# Input: [100]
# Output: 1# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxTurbulenceSize(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        result = 1
        start = 0
        for i in xrange(1, len(A)):
            if i == len(A)-1 or \
               cmp(A[i-1], A[i]) * cmp(A[i], A[i+1]) != -1:
                result = max(result, i-start+1)
                start = i
        return result
```

next-greater-node-in-linked-list.py

```python
# Example 2:
#
# Input: [2,7,4,3,5]
# Output: [7,0,5,5,0]
#
#
#
# Example 3:
#
# Input: [1,7,5,1,9,2,5,1]
# Output: [7,9,9,9,0,5,0,0]
#
#
#
#
# Note:
#
#
#       1 <= node.val <= 10^9 for each node in the linked list.
#       The given list has length in the range [0, 10000].# Time:  O(n)
# Space: O(n)

# Definition for singly-linked list.
class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None


class Solution(object):
    def nextLargerNodes(self, head):
        """
        :type head: ListNode
        :rtype: List[int]
        """
        result, stk = [], []
        while head:
            while stk and stk[-1][1] < head.val:
                result[stk.pop()[0]] = head.val
            stk.append([len(result), head.val])
            result.append(0)
            head = head.next
        return result
```

friend-circles.py

```
# There are N students in a class. Some of them are friends, while some are not.
# Their friendship is transitive in nature. For example, if A is a direct friend
# of B, and B is a direct friend of C, then A is an indirect friend of C. And we
# defined a friend circle is a group of students who are direct or indirect
# friends.
#
# Given a N*N matrix M representing the friend relationship between students in
# the class. If M[i][j] = 1, then the ith and jth students are direct friends with
# each other, otherwise not. And you have to output the total number of friend
# circles among all the students.
#
# Example 1:
#
# Input:
# [[1,1,0],
#  [1,1,0],
#  [0,0,1]]
# Output: 2
# Explanation:The 0th and 1st students are direct friends, so they are in a
# friend circle.
# The 2nd student himself is in a friend circle. So return 2.
#
#
#
#
# Example 2:
#
# Input:
# [[1,1,0],
#  [1,1,1],
#  [0,1,1]]
# Output: 1
# Explanation:The 0th and 1st students are direct friends, the 1st and 2nd
# students are direct friends,
# so the 0th and 2nd students are indirect friends. All of them are in the same
# friend circle, so return 1.
#
#
#
#
# Constraints:
#
#
#       1 <= N <= 200
#       M[i][i] == 1
#       M[i][j] == M[j][i]# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def findCircleNum(self, M):
        """
        :type M: List[List[int]]
        :rtype: int
        """

        class UnionFind(object):
            def __init__(self, n):
                self.set = range(n)
```

```
        self.count = n

    def find_set(self, x):
       if self.set[x] != x:
           self.set[x] = self.find_set(self.set[x])  # path compression.
       return self.set[x]

    def union_set(self, x, y):
        x_root, y_root = map(self.find_set, (x, y))
        if x_root != y_root:
            self.set[min(x_root, y_root)] = max(x_root, y_root)
            self.count -= 1

circles = UnionFind(len(M))
for i in xrange(len(M)):
    for j in xrange(len(M)):
        if M[i][j] and i != j:
            circles.union_set(i, j)
return circles.count
```

score-of-parentheses.py

```
# Given a balanced parentheses string S, compute the score of the string based
# on the following rule:
#
#
#       () has score 1
#       AB has score A + B, where A and B are balanced parentheses strings.
#       (A) has score 2 * A, where A is a balanced parentheses string.
#
#
#
#
#
# Example 1:
#
# Input: "()"
# Output: 1
#
#
#
# Example 2:
#
# Input: "(())"
# Output: 2
#
#
#
# Example 3:
#
# Input: "()()"
# Output: 2
#
#
#
# Example 4:
#
# Input: "(()(()))"
# Output: 6
#
#
#
#
# Note:
#
#
#       S is a balanced parentheses string, containing only ( and ).
#       2 <= S.length <= 50# Time:  O(n)
# Space: O(1)


class Solution(object):
    def scoreOfParentheses(self, S):
        """
        :type S: str
        :rtype: int
        """
        result, depth = 0, 0
        for i in xrange(len(S)):
            if S[i] == '(':
```

```python
                depth += 1
            else:
                depth -= 1
                if S[i-1] == '(':
                    result += 2**depth
        return result


# Time:  O(n)
# Space: O(h)
class Solution2(object):
    def scoreOfParentheses(self, S):
        """
        :type S: str
        :rtype: int
        """
        stack = [0]
        for c in S:
            if c == '(':
                stack.append(0)
            else:
                last = stack.pop()
                stack[-1] += max(1, 2*last)
        return stack[0]
```

beautiful-arrangement-ii.py

```python
# Given two integers n and k, you need to construct a list which contains n
# different positive integers ranging from 1 to n and obeys the following
# requirement:
#
#
# Suppose this list is [a1, a2, a3, ... , an], then the list [|a1 - a2|, |a2 -
# a3|, |a3 - a4|, ... , |an-1 - an|] has exactly k distinct integers.
#
#
#
# If there are multiple answers, print any of them.
#
#
# Example 1:
#
# Input: n = 3, k = 1
# Output: [1, 2, 3]
# Explanation: The [1, 2, 3] has three different positive integers ranging from
# 1 to 3, and the [1, 1] has exactly 1 distinct integer: 1.
#
#
#
# Example 2:
#
# Input: n = 3, k = 2
# Output: [1, 3, 2]
# Explanation: The [1, 3, 2] has three different positive integers ranging from
# 1 to 3, and the [2, 1] has exactly 2 distinct integers: 1 and 2.
#
#
#
# Note:
#
#
# The n and k are in the range 1 <= k < n <= 104.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def constructArray(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: List[int]
        """
        result = []
        left, right = 1, n
        while left <= right:
            if k % 2:
                result.append(left)
                left += 1
            else:
                result.append(right)
                right -= 1
            if k > 1:
                k -= 1
        return result
```

matrix-block-sum.py

```python
# Given a m * n matrix mat and an integer K, return a matrix answer where each
# answer[i][j] is the sum of all elements mat[r][c] for i - K <= r <= i + K, j - K
# <= c <= j + K, and (r, c) is a valid position in the matrix.
#
# Example 1:
#
# Input: mat = [[1,2,3],[4,5,6],[7,8,9]], K = 1
# Output: [[12,21,16],[27,45,33],[24,39,28]]
#
#
# Example 2:
#
# Input: mat = [[1,2,3],[4,5,6],[7,8,9]], K = 2
# Output: [[45,45,45],[45,45,45],[45,45,45]]
#
#
#
# Constraints:
#
#
#       m == mat.length
#       n == mat[i].length
#       1 <= m, n, K <= 100
#       1 <= mat[i][j] <= 100# Time:  O(m * n)
# Space: O(m * n)

class Solution(object):
    def matrixBlockSum(self, mat, K):
        """
        :type mat: List[List[int]]
        :type K: int
        :rtype: List[List[int]]
        """
        m, n = len(mat), len(mat[0])
        accu = [[0 for _ in xrange(n+1)] for _ in xrange(m+1)]
        for i in xrange(m):
            for j in xrange(n):
                accu[i+1][j+1] = accu[i+1][j]+accu[i][j+1]-accu[i][j]+mat[i][j]
        result = [[0 for _ in xrange(n)] for _ in xrange(m)]
        for i in xrange(m):
            for j in xrange(n):
                r1, c1, r2, c2 = max(i-K, 0), max(j-K, 0), min(i+K+1, m), min(j+K+1, n)
                result[i][j] = accu[r2][c2]-accu[r1][c2]-accu[r2][c1]+accu[r1][c1]
        return result
```

maximum-of-absolute-value-expression.py

```
# Given two arrays of integers with equal lengths, return the maximum value of:
#
# |arr1[i] - arr1[j]| + |arr2[i] - arr2[j]| + |i - j|
#
# where the maximum is taken over all 0 <= i, j < arr1.length.
#
#
# Example 1:
#
# Input: arr1 = [1,2,3,4], arr2 = [-1,4,5,6]
# Output: 13
#
#
# Example 2:
#
# Input: arr1 = [1,-2,-5,0,10], arr2 = [0,-2,-1,-7,-4]
# Output: 20
#
#
#
# Constraints:
#
#
#       2 <= arr1.length == arr2.length <= 40000
#       -10^6 <= arr1[i], arr2[i] <= 10^6# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxAbsValExpr(self, arr1, arr2):
        """
        :type arr1: List[int]
        :type arr2: List[int]
        :rtype: int
        """
        # 1. max(|arr1[i]-arr1[j]| + |arr2[i]-arr2[j]| + |i-j| for i > j)
        #    = max(|arr1[i]-arr1[j]| + |arr2[i]-arr2[j]| + |i-j| for j > i)
        # 2. for i > j:
        #         (|arr1[i]-arr1[j]| + |arr2[i]-arr2[j]| + |i-j|)
        #         >= c1*(arr1[i]-arr1[j]) + c2*(arr2[i]-arr2[j]) + i-j for c1 in (1, -1), c2 in (1, -1)
        #         = (c1*arr1[i]+c2*arr2[i]+i) - (c1*arr1[j]+c2*arr2[j]+j) for c1 in (1, -1), c2 in (1, -1)
        # 1 + 2 => max(|arr1[i]-arr1[j]| + |arr2[i]-arr2[j]| + |i-j| for i != j)
        #           = max((c1*arr1[i]+c2*arr2[i]+i) - (c1*arr1[j]+c2*arr2[j]+j)
        #                  for c1 in (1, -1), c2 in (1, -1) for i > j)
        result = 0
        for c1 in [1, -1]:
            for c2 in [1, -1]:
                min_prev = float("inf")
                for i in xrange(len(arr1)):
                    curr = c1*arr1[i] + c2*arr2[i] + i
                    result = max(result, curr-min_prev)
                    min_prev = min(min_prev, curr)
        return result


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def maxAbsValExpr(self, arr1, arr2):
```

```python
"""
:type arr1: List[int]
:type arr2: List[int]
:rtype: int
"""
return max(max(c1*arr1[i] + c2*arr2[i] + i for i in xrange(len(arr1))) -
           min(c1*arr1[i] + c2*arr2[i] + i for i in xrange(len(arr1)))
           for c1 in [1, -1] for c2 in [1, -1])
```

single-element-in-a-sorted-array.py

```python
# You are given a sorted array consisting of only integers where every element
# appears exactly twice, except for one element which appears exactly once. Find
# this single element that appears only once.
#
# Follow up: Your solution should run in O(log n) time and O(1) space.
#
#
# Example 1:
# Input: nums = [1,1,2,3,3,4,4,8,8]
# Output: 2
# Example 2:
# Input: nums = [3,3,7,7,10,11,11]
# Output: 10
#
#
# Constraints:
#
#
#       1 <= nums.length <= 10^5
#       0 <= nums[i] <= 10^5# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def singleNonDuplicate(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        left, right = 0, len(nums)-1
        while left <= right:
            mid = left + (right - left) / 2
            if not (mid%2 == 0 and mid+1 < len(nums) and \
                    nums[mid] == nums[mid+1]) and \
               not (mid%2 == 1 and nums[mid] == nums[mid-1]):
                right = mid-1
            else:
                left = mid+1
        return nums[left]

    def singleNonDuplicate2(self, nums):
        # odd xor 1 = odd-1
        # even xor 1 = even+1

        lo, hi = 0, len(nums) - 1
        while lo < hi:
            mid = (lo + hi) / 2
            if nums[mid] == nums[mid ^ 1]:
                lo = mid + 1
            else:
                hi = mid
        return nums[lo]
```

group-anagrams.py

```python
# Given an array of strings strs, group the anagrams together. You can return
# the answer in any order.
#
# An Anagram is a word or phrase formed by rearranging the letters of a
# different word or phrase, typically using all the original letters exactly once.
#
#
# Example 1:
# Input: strs = ["eat","tea","tan","ate","nat","bat"]
# Output: [["bat"],["nat","tan"],["ate","eat","tea"]]
# Example 2:
# Input: strs = [""]
# Output: [[""]]
# Example 3:
# Input: strs = ["a"]
# Output: [["a"]]
#
#
# Constraints:
#
#
#       1 <= strs.length <= 104
#       0 <= strs[i].length <= 100
#       strs[i] consists of lower-case English letters.# Time:  O(n * glogg), g is the max size of groups.
# Space: O(n)

import collections


class Solution(object):
    def groupAnagrams(self, strs):
        """
        :type strs: List[str]
        :rtype: List[List[str]]
        """
        anagrams_map, result = collections.defaultdict(list), []
        for s in strs:
            sorted_str = ("").join(sorted(s))
            anagrams_map[sorted_str].append(s)
        for anagram in anagrams_map.values():
            anagram.sort()
            result.append(anagram)
        return result
```

uncrossed-lines.py

```python
# We write the integers of A and B (in the order they are given) on two separate
# horizontal lines.
#
# Now, we may draw connecting lines: a straight line connecting two numbers A[i]
# and B[j] such that:
#
#
#       A[i] == B[j];
#       The line we draw does not intersect any other connecting (non-
# horizontal) line.
#
#
# Note that a connecting lines cannot intersect even at the endpoints: each
# number can only belong to one connecting line.
#
# Return the maximum number of connecting lines we can draw in this way.
#
#
#
# Example 1:
#
# Input: A = [1,4,2], B = [1,2,4]
# Output: 2
# Explanation: We can draw 2 uncrossed lines as in the diagram.
# We cannot draw 3 uncrossed lines, because the line from A[1]=4 to B[2]=4 will
# intersect the line from A[2]=2 to B[1]=2.
#
#
#
# Example 2:
#
# Input: A = [2,5,1,2,5], B = [10,5,2,1,5,2]
# Output: 3
#
#
#
# Example 3:
#
# Input: A = [1,3,7,1,7,5], B = [1,9,2,5,1]
# Output: 2
#
#
#
#
#
# Note:
#
#
#       1 <= A.length <= 500
#       1 <= B.length <= 500
#       1 <= A[i], B[i] <= 2000# Time:  O(m * n)
# Space: O(min(m, n))

class Solution(object):
    def maxUncrossedLines(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
```

```python
        :rtype: int
        """
        if len(A) < len(B):
            return self.maxUncrossedLines(B, A)

        dp = [[0 for _ in xrange(len(B)+1)] for _ in xrange(2)]
        for i in xrange(len(A)):
            for j in xrange(len(B)):
                dp[(i+1)%2][j+1] = max(dp[i%2][j] + int(A[i] == B[j]),
                                       dp[i%2][j+1],
                                       dp[(i+1)%2][j])
        return dp[len(A)%2][len(B)]
```

partition-list.py

```python
# 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.
#
# Example:
#
# Input: head = 1->4->3->2->5->2, x = 3
# Output: 1->2->2->4->3->5# Time:  O(n)
# Space: O(1)

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

    def __repr__(self):
        if self:
            return "{} -> {}".format(self.val, repr(self.next))

class Solution(object):
    # @param head, a ListNode
    # @param x, an integer
    # @return a ListNode
    def partition(self, head, x):
        dummySmaller, dummyGreater = ListNode(-1), ListNode(-1)
        smaller, greater = dummySmaller, dummyGreater

        while head:
            if head.val < x:
                smaller.next = head
                smaller = smaller.next
            else:
                greater.next = head
                greater = greater.next
            head = head.next

        smaller.next = dummyGreater.next
        greater.next = None

        return dummySmaller.next
```

filling-bookcase-shelves.py

```python
# We have a sequence of books: the i-th book has thickness books[i][0] and
# height books[i][1].
#
# We want to place these books in order onto bookcase shelves that have total
# width shelf_width.
#
# We choose some of the books to place on this shelf (such that the sum of their
# thickness is <= shelf_width), then build another level of shelf of the bookcase
# so that the total height of the bookcase has increased by the maximum height of
# the books we just put down.  We repeat this process until there are no more
# books to place.
#
# Note again that at each step of the above process, the order of the books we
# place is the same order as the given sequence of books.  For example, if we have
# an ordered list of 5 books, we might place the first and second book onto the
# first shelf, the third book on the second shelf, and the fourth and fifth book
# on the last shelf.
#
# Return the minimum possible height that the total bookshelf can be after
# placing shelves in this manner.
#
#
# Example 1:
#
# Input: books = [[1,1],[2,3],[2,3],[1,1],[1,1],[1,1],[1,2]], shelf_width = 4
# Output: 6
# Explanation:
# The sum of the heights of the 3 shelves are 1 + 3 + 2 = 6.
# Notice that book number 2 does not have to be on the first shelf.
#
#
#
# Constraints:
#
#
#       1 <= books.length <= 1000
#       1 <= books[i][0] <= shelf_width <= 1000
#       1 <= books[i][1] <= 1000# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def minHeightShelves(self, books, shelf_width):
        """
        :type books: List[List[int]]
        :type shelf_width: int
        :rtype: int
        """
        dp = [float("inf") for _ in xrange(len(books)+1)]
        dp[0] = 0
        for i in xrange(1, len(books)+1):
            max_width = shelf_width
            max_height = 0
            for j in reversed(xrange(i)):
                if max_width-books[j][0] < 0:
                    break
                max_width -= books[j][0]
                max_height = max(max_height, books[j][1])
                dp[i] = min(dp[i], dp[j]+max_height)
```

```python
    return dp[len(books)]
```

sum-of-mutated-array-closest-to-target.py

```python
# Given an integer array arr and a target value target, return the
# integer value such that when we change all the integers larger than value in the
# given array to be equal to value, the sum of the array gets as close as possible
# (in absolute difference) to target.
#
# In case of a tie, return the minimum such integer.
#
# Notice that the answer is not neccesarilly a number from arr.
#
#
# Example 1:
#
# Input: arr = [4,9,3], target = 10
# Output: 3
# Explanation: When using 3 arr converts to [3, 3, 3] which sums 9 and that's
# the optimal answer.
#
#
# Example 2:
#
# Input: arr = [2,3,5], target = 10
# Output: 5
#
#
# Example 3:
#
# Input: arr = [60864,25176,27249,21296,20204], target = 56803
# Output: 11361
#
#
#
# Constraints:
#
#
#       1 <= arr.length <= 10^4
#       1 <= arr[i], target <= 10^5# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def findBestValue(self, arr, target):
        """
        :type arr: List[int]
        :type target: int
        :rtype: int
        """
        arr.sort(reverse=True)
        max_arr = arr[0]
        while arr and arr[-1]*len(arr) <= target:
            target -= arr.pop()
        # let x = ceil(t/n)-1
        # (1) (t/n-1/2) <= x:
        #    return x, which is equal to ceil(t/n)-1 = ceil(t/n-1/2) = (2t+n-1)//2n
        # (2) (t/n-1/2) > x:
        #    return x+1, which is equal to ceil(t/n) = ceil(t/n-1/2) = (2t+n-1)//2n
        # (1) + (2) => both return (2t+n-1)//2n
        return max_arr if not arr else (2*target+len(arr)-1)//(2*len(arr))
```

```python
# Time:  O(nlogn)
# Space: O(1)
class Solution2(object):
    def findBestValue(self, arr, target):
        """
        :type arr: List[int]
        :type target: int
        :rtype: int
        """
        arr.sort(reverse=True)
        max_arr = arr[0]
        while arr and arr[-1]*len(arr) <= target:
            target -= arr.pop()
        if not arr:
            return max_arr
        x = (target-1)//len(arr)
        return x if target-x*len(arr) <= (x+1)*len(arr)-target else x+1


# Time:  O(nlogm), m is the max of arr, which may be larger than n
# Space: O(1)
class Solution3(object):
    def findBestValue(self, arr, target):
        """
        :type arr: List[int]
        :type target: int
        :rtype: int
        """
        def total(arr, v):
            result = 0
            for x in arr:
                result += min(v, x)
            return result

        def check(arr, v, target):
            return total(arr, v) >= target

        left, right = 1, max(arr)
        while left <= right:
            mid = left + (right-left)//2
            if check(arr, mid, target):
                right = mid-1
            else:
                left = mid+1
        return left-1 if target-total(arr, left-1) <= total(arr, left)-target else left
```

2-keys-keyboard.py

```python
# Initially on a notepad only one character 'A' is present. You can perform two
# operations on this notepad for each step:
#
#
#       Copy All: You can copy all the characters present on the notepad
# (partial copy is not allowed).
#       Paste: You can paste the characters which are copied last time.
#
#
#
#
# Given a number n. You have to get exactly n 'A' on the notepad by performing
# the minimum number of steps permitted. Output the minimum number of steps to get
# n 'A'.
#
# Example 1:
#
# Input: 3
# Output: 3
# Explanation:
# Intitally, we have one character 'A'.
# In step 1, we use Copy All operation.
# In step 2, we use Paste operation to get 'AA'.
# In step 3, we use Paste operation to get 'AAA'.
#
#
#
#
# Note:
#
#
#       The n will be in the range [1, 1000].# Time:  O(sqrt(n))
# Space: O(1)

class Solution(object):
    def minSteps(self, n):
        """
        :type n: int
        :rtype: int
        """
        result = 0
        p = 2
        # the answer is the sum of prime factors
        while p**2 <= n:
            while n % p == 0:
                result += p
                n //= p
            p += 1
        if n > 1:
            result += n
        return result
```

binary-search-tree-iterator.py

```python
# 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.
#
#
#
#
#
#
# Example:
#
#
#
# BSTIterator iterator = new BSTIterator(root);
# iterator.next();    // return 3
# iterator.next();    // return 7
# iterator.hasNext(); // return true
# iterator.next();    // return 9
# iterator.hasNext(); // return true
# iterator.next();    // return 15
# iterator.hasNext(); // return true
# iterator.next();    // return 20
# iterator.hasNext(); // return false
#
#
#
#
# Note:
#
#
#       next() and hasNext() should run in average O(1) time and uses O(h)
# memory, where h is the height of the tree.
#       You may assume that next() call will always be valid, that is, there
# will be at least a next smallest number in the BST when next() is called.# Time:  O(1)
# Space: O(h), h is height of binary tree

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class BSTIterator(object):
    # @param root, a binary search tree's root node
    def __init__(self, root):
        self.stack = []
        self.cur = root

    # @return a boolean, whether we have a next smallest number
    def hasNext(self):
        return self.stack or self.cur

    # @return an integer, the next smallest number
    def next(self):
        while self.cur:
            self.stack.append(self.cur)
```

```
        self.cur = self.cur.left

    self.cur = self.stack.pop()
    node = self.cur
    self.cur = self.cur.right

    return node.val
```

find-the-duplicate-number.py

```
# 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.
#
# Example 1:
#
# Input: [1,3,4,2,2]
# Output: 2
#
#
# Example 2:
#
# Input: [3,1,3,4,2]
# Output: 3
#
# 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.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findDuplicate(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        # Treat each (key, value) pair of the array as the (pointer, next) node of the linked list,
        # thus the duplicated number will be the begin of the cycle in the linked list.
        # Besides, there is always a cycle in the linked list which
        # starts from the first element of the array.
        slow = nums[0]
        fast = nums[nums[0]]
        while slow != fast:
            slow = nums[slow]
            fast = nums[nums[fast]]

        fast = 0
        while slow != fast:
            slow = nums[slow]
            fast = nums[fast]
        return slow


# Time:  O(nlogn)
# Space: O(1)
# Binary search method.
class Solution2(object):
    def findDuplicate(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        left, right = 1, len(nums) - 1
```

```python
        while left <= right:
            mid = left + (right - left) / 2
            # Get count of num <= mid.
            count = 0
            for num in nums:
                if num <= mid:
                    count += 1
            if count > mid:
                right = mid - 1
            else:
                left = mid + 1
        return left

# Time:  O(n)
# Space: O(n)
class Solution3(object):
    def findDuplicate(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        duplicate = 0
        # Mark the value as visited by negative.
        for num in nums:
            if nums[abs(num) - 1] > 0:
                nums[abs(num) - 1] *= -1
            else:
                duplicate = abs(num)
                break
        # Rollback the value.
        for num in nums:
            if nums[abs(num) - 1] < 0:
                nums[abs(num) - 1] *= -1
            else:
                break
        return duplicate
```

merge-intervals.py

```python
# Given a collection of intervals, merge all overlapping intervals.
#
# Example 1:
#
# Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
# Output: [[1,6],[8,10],[15,18]]
# Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
#
#
# Example 2:
#
# Input: intervals = [[1,4],[4,5]]
# Output: [[1,5]]
# Explanation: Intervals [1,4] and [4,5] are considered overlapping.
#
# NOTE: input types have been changed on April 15, 2019. Please reset to default
# code definition to get new method signature.
#
#
# Constraints:
#
#
#       intervals[i][0] <= intervals[i][1]# Time:  O(nlogn)
# Space: O(1)

class Interval(object):
    def __init__(self, s=0, e=0):
        self.start = s
        self.end = e

    def __repr__(self):
        return "[{}, {}]".format(self.start, self.end)


class Solution(object):
    def merge(self, intervals):
        """
        :type intervals: List[Interval]
        :rtype: List[Interval]
        """
        if not intervals:
            return intervals

        intervals.sort(key=lambda x: x.start)
        iterator = iter(intervals)
        result = [next(iterator)]
        for current in iterator:
            prev = result[-1]
            if current.start <= prev.end:
                prev.end = max(current.end, prev.end)
            else:
                result.append(current)

        return result
```

largest-values-from-labels.py

```python
# Example 1:
#
# Input: values = [5,4,3,2,1], labels = [1,1,2,2,3], num_wanted = 3, use_limit =
# 1
# Output: 9
# Explanation: The subset chosen is the first, third, and fifth item.
#
#
#
# Example 2:
#
# Input: values = [5,4,3,2,1], labels = [1,3,3,3,2], num_wanted = 3, use_limit =
# 2
# Output: 12
# Explanation: The subset chosen is the first, second, and third item.
#
#
#
# Example 3:
#
# Input: values = [9,8,8,7,6], labels = [0,0,0,1,1], num_wanted = 3, use_limit =
# 1
# Output: 16
# Explanation: The subset chosen is the first and fourth item.
#
#
#
# Example 4:
#
# Input: values = [9,8,8,7,6], labels = [0,0,0,1,1], num_wanted = 3, use_limit =
# 2
# Output: 24
# Explanation: The subset chosen is the first, second, and fourth item.
#
#
#
#
# Note:
#
#
#       1 <= values.length == labels.length <= 20000
#       0 <= values[i], labels[i] <= 20000
#       1 <= num_wanted, use_limit <= values.length# Time:  O(nlogn)
# Space: O(n)

import collections


class Solution(object):
    def largestValsFromLabels(self, values, labels, num_wanted, use_limit):
        """
        :type values: List[int]
        :type labels: List[int]
        :type num_wanted: int
        :type use_limit: int
        :rtype: int
        """
        counts = collections.defaultdict(int)
```

```python
val_labs = zip(values,labels)
val_labs.sort(reverse=True)
result = 0
for val, lab in val_labs:
    if counts[lab] >= use_limit:
        continue
    result += val
    counts[lab] += 1
    num_wanted -= 1
    if num_wanted == 0:
        break
return result
```

3sum-closest.py

```python
# Given an array nums of n integers and an integer target, find three integers
# in nums such that the sum is closest to target. Return the sum of the three
# integers. You may assume that each input would have exactly one solution.
#
#
# Example 1:
#
# Input: nums = [-1,2,1,-4], target = 1
# Output: 2
# Explanation: The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
#
#
#
# Constraints:
#
#
#       3 <= nums.length <= 10^3
#       -10^3 <= nums[i] <= 10^3
#       -10^4 <= target <= 10^4# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    def threeSumClosest(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        nums, result, min_diff, i = sorted(nums), float("inf"), float("inf"), 0
        while i < len(nums) - 2:
            if i == 0 or nums[i] != nums[i - 1]:
                j, k = i + 1, len(nums) - 1
                while j < k:
                    diff = nums[i] + nums[j] + nums[k] - target
                    if abs(diff) < min_diff:
                        min_diff = abs(diff)
                        result = nums[i] + nums[j] + nums[k]
                    if diff < 0:
                        j += 1
                    elif diff > 0:
                        k -= 1
                    else:
                        return target
            i += 1
        return result
```

solve-the-equation.py

```
# Solve a given equation and return the value of x in the form of string
# "x=#value". The equation contains only '+', '-' operation, the variable x and
# its coefficient.
#
#
#
# If there is no solution for the equation, return "No solution".
#
#
# If there are infinite solutions for the equation, return "Infinite solutions".
#
#
# If there is exactly one solution for the equation, we ensure that the value of
# x is an integer.
#
#
# Example 1:
#
# Input: "x+5-3+x=6+x-2"
# Output: "x=2"
#
#
#
# Example 2:
#
# Input: "x=x"
# Output: "Infinite solutions"
#
#
#
# Example 3:
#
# Input: "2x=x"
# Output: "x=0"
#
#
#
# Example 4:
#
# Input: "2x+3x-6x=x+2"
# Output: "x=-1"
#
#
#
# Example 5:
#
# Input: "x=x+2"
# Output: "No solution"# Time:  O(n)
# Space: O(n)

import re


class Solution(object):
    def solveEquation(self, equation):
        """
        :type equation: str
        :rtype: str
```

```
"""
a, b, side = 0, 0, 1
for eq, sign, num, isx in re.findall('(=)|([-+]?)(\d*)(x?)', equation):
    if eq:
        side = -1
    elif isx:
        a += side * int(sign + '1') * int(num or 1)
    elif num:
        b -= side * int(sign + num)
return 'x=%d' % (b / a) if a else 'No solution' if b else 'Infinite solutions'
```

minimum-height-trees.py

```python
# For an 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 :
#
# Input: n = 4, edges = [[1, 0], [1, 2], [1, 3]]
#
#         0
#         |
#         1
#        / \
#       2   3
#
# Output: [1]
#
#
# Example 2 :
#
# Input: n = 6, edges = [[0, 3], [1, 3], [2, 3], [4, 3], [5, 4]]
#
#      0  1  2
#       \ | /
#         3
#         |
#         4
#         |
#         5
#
# Output: [3, 4]
#
# Note:
#
#
#       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."
#       The height of a rooted tree is the number of edges on the longest
# downward path between the root and a leaf.# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def findMinHeightTrees(self, n, edges):
```

```python
"""
:type n: int
:type edges: List[List[int]]
:rtype: List[int]
"""
if n == 1:
    return [0]

neighbors = collections.defaultdict(set)
for u, v in edges:
    neighbors[u].add(v)
    neighbors[v].add(u)

pre_level, unvisited = [], set()
for i in xrange(n):
    if len(neighbors[i]) == 1:  # A leaf.
        pre_level.append(i)
    unvisited.add(i)

# A graph can have 2 MHTs at most.
# BFS from the leaves until the number
# of the unvisited nodes is less than 3.
while len(unvisited) > 2:
    cur_level = []
    for u in pre_level:
        unvisited.remove(u)
        for v in neighbors[u]:
            if v in unvisited:
                neighbors[v].remove(u)
                if len(neighbors[v]) == 1:
                    cur_level.append(v)
    pre_level = cur_level

return list(unvisited)
```

continuous-subarray-sum.py

```python
# Given a list of non-negative numbers and a target integer k, write a function
# to check if the array has a continuous subarray of size at least 2 that sums up
# to a multiple of k, that is, sums up to n*k where n is also an integer.
#
#
#
# Example 1:
#
# Input: [23, 2, 4, 6, 7],  k=6
# Output: True
# Explanation: Because [2, 4] is a continuous subarray of size 2 and sums up to
# 6.
#
#
# Example 2:
#
# Input: [23, 2, 6, 4, 7],  k=6
# Output: True
# Explanation: Because [23, 2, 6, 4, 7] is an continuous subarray of size 5 and
# sums up to 42.
#
#
#
# Constraints:
#
#
#       The length of the array won't exceed 10,000.
#       You may assume the sum of all the numbers is in the range of a signed
# 32-bit integer.# Time:  O(n)
# Space: O(k)

class Solution(object):
    def checkSubarraySum(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: bool
        """
        count = 0
        lookup = {0: -1}
        for i, num in enumerate(nums):
            count += num
            if k:
                count %= k
            if count in lookup:
                if i - lookup[count] > 1:
                    return True
            else:
                lookup[count] = i

        return False
```

brick-wall.py

```python
# There is a brick wall in front of you. The wall is rectangular and has several
# rows of bricks. The bricks have the same height but different width. You want to
# draw a vertical line from the top to the bottom and cross the least bricks.
#
# The brick wall is represented by a list of rows. Each row is a list of
# integers representing the width of each brick in this row from left to right.
#
# If your line go through the edge of a brick, then the brick is not considered
# as crossed. You need to find out how to draw the line to cross the least bricks
# and return the number of crossed bricks.
#
# You cannot draw a line just along one of the two vertical edges of the wall,
# in which case the line will obviously cross no bricks.
#
#
#
# Example:
#
# Input: [[1,2,2,1],
#         [3,1,2],
#         [1,3,2],
#         [2,4],
#         [3,1,2],
#         [1,3,1,1]]
#
# Output: 2
#
# Explanation:
#
#
#
#
#
# Note:
#
#
#       The width sum of bricks in different rows are the same and won't exceed
# INT_MAX.
#       The number of bricks in each row is in range [1,10,000]. The height of
# wall is in range [1,10,000]. Total number of bricks of the wall won't exceed
# 20,000.# Time:  O(n), n is the total number of the bricks
# Space: O(m), m is the total number different widths

import collections


class Solution(object):
    def leastBricks(self, wall):
        """
        :type wall: List[List[int]]
        :rtype: int
        """
        widths = collections.defaultdict(int)
        result = len(wall)
        for row in wall:
            width = 0
            for i in xrange(len(row)-1):
                width += row[i]
```

```
        widths[width] += 1
        result = min(result, len(wall) - widths[width])
return result
```

gray-code.py

```python
# 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.
#
# Example 1:
#
# Input: 2
# Output: [0,1,3,2]
# Explanation:
# 00 - 0
# 01 - 1
# 11 - 3
# 10 - 2
#
# For a given n, a gray code sequence may not be uniquely defined.
# For example, [0,2,3,1] is also a valid gray code sequence.
#
# 00 - 0
# 10 - 2
# 11 - 3
# 01 - 1
#
#
# Example 2:
#
# Input: 0
# Output: [0]
# Explanation: We define the gray code sequence to begin with 0.
#              A gray code sequence of n has size = 2n, which for n = 0 the size
# is 20 = 1.
#              Therefore, for n = 0 the gray code sequence is [0].# Time:  O(2^n)
# Space: O(1)

class Solution(object):
    def grayCode(self, n):
        """
        :type n: int
        :rtype: List[int]
        """
        result = [0]
        for i in xrange(n):
            for n in reversed(result):
                result.append(1 << i | n)
        return result


# Proof of closed form formula could be found here:
# http://math.stackexchange.com/questions/425894/proof-of-closed-form-formula-to-convert-a-binary-number-to-it
class Solution2(object):
    def grayCode(self, n):
        """
        :type n: int
        :rtype: List[int]
        """
        return [i >> 1 ^ i for i in xrange(1 << n)]
```

odd-even-linked-list.py

```python
# 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 1:
#
# Input: 1->2->3->4->5->NULL
# Output: 1->3->5->2->4->NULL
#
#
# Example 2:
#
# Input: 2->1->3->5->6->4->7->NULL
# Output: 2->3->6->7->1->5->4->NULL
#
#
#
# Constraints:
#
#
#       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 ...
#       The length of the linked list is between [0, 10^4].# Time:  O(n)
# Space: O(1)

class Solution(object):
    def oddEvenList(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if head:
            odd_tail, cur = head, head.next
            while cur and cur.next:
                even_head = odd_tail.next
                odd_tail.next = cur.next
                odd_tail = odd_tail.next
                cur.next = odd_tail.next
                odd_tail.next = even_head
                cur = cur.next
        return head

def oddEvenList(self, head):
    dummy1 = odd = ListNode(0)
    dummy2 = even = ListNode(0)
    while head:
        odd.next = head
        even.next = head.next
        odd = odd.next
        even = even.next
        head = head.next.next if even else None
    odd.next = dummy2.next
    return dummy1.next```
newpage
```

longest-uncommon-subsequence-ii.py

```python
# Given a list of strings, you need to find the longest uncommon subsequence
# among them. The longest uncommon subsequence is defined as the longest
# subsequence of one of these strings and this subsequence should not be any
# subsequence of the other strings.
#
#
#
# A subsequence is a sequence that can be derived from one sequence by deleting
# some characters without changing the order of the remaining elements. Trivially,
# any string is a subsequence of itself and an empty string is a subsequence of
# any string.
#
#
#
# The input will be a list of strings, and the output needs to be the length of
# the longest uncommon subsequence. If the longest uncommon subsequence doesn't
# exist, return -1.
#
#
# Example 1:
#
# Input: "aba", "cdc", "eae"
# Output: 3
#
#
#
# Note:
#
# All the given strings' lengths will not exceed 10.
# The length of the given list will be in the range of [2, 50].# Time:  O(l * n^2)
# Space: O(1)

class Solution(object):
    def findLUSlength(self, A):
        def subseq(w1, w2):
            # True iff word1 is a subsequence of word2.
            i = 0
            for c in w2:
                if i < len(w1) and w1[i] == c:
                    i += 1
            return i == len(w1)

        A.sort(key=len, reverse=True)
        for i, word1 in enumerate(A):
            if all(not subseq(word1, word2)
                   for j, word2 in enumerate(A) if i != j):
                return len(word1)
        return -1
```
newpage

vertical-order-traversal-of-a-binary-tree.py

```python
# Example 2:
#
#
#
# Input: [1,2,3,4,5,6,7]
```

```python
# Output: [[4],[2],[1,5,6],[3],[7]]
# Explanation:
# The node with value 5 and the node with value 6 have the same position
# according to the given scheme.
# However, in the report "[1,5,6]", the node value of 5 comes first since 5 is
# smaller than 6.# Time:  O(nlogn)
# Space: O(n)

import collections


# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def verticalTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        def dfs(node, lookup, x, y):
            if not node:
                return
            lookup[x][y].append(node)
            dfs(node.left, lookup, x-1, y+1)
            dfs(node.right, lookup, x+1, y+1)

        lookup = collections.defaultdict(lambda: collections.defaultdict(list))
        dfs(root, lookup, 0, 0)

        result = []
        for x in sorted(lookup):
            report = []
            for y in sorted(lookup[x]):
                report.extend(sorted(node.val for node in lookup[x][y]))
            result.append(report)
        return result
```

basic-calculator-ii.py

```python
# 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.
#
# Example 1:
#
# Input: "3+2*2"
# Output: 7
#
#
# Example 2:
#
# Input: " 3/2 "
# Output: 1
#
# Example 3:
#
# Input: " 3+5 / 2 "
# Output: 5
#
#
# Note:
#
#
#       You may assume that the given expression is always valid.
#       Do not use the eval built-in library function.# Time:  O(n)
# Space: O(n)


class Solution(object):
    # @param {string} s
    # @return {integer}
    def calculate(self, s):
        operands, operators = [], []
        operand = ""
        for i in reversed(xrange(len(s))):
            if s[i].isdigit():
                operand += s[i]
                if i == 0 or not s[i-1].isdigit():
                    operands.append(int(operand[::-1]))
                    operand = ""
            elif s[i] == ')' or s[i] == '*' or s[i] == '/':
                operators.append(s[i])
            elif s[i] == '+' or s[i] == '-':
                while operators and \
                     (operators[-1] == '*' or operators[-1] == '/'):
                    self.compute(operands, operators)
                operators.append(s[i])
            elif s[i] == '(':
                while operators[-1] != ')':
                    self.compute(operands, operators)
                operators.pop()

        while operators:
            self.compute(operands, operators)

        return operands[-1]
```

```python
def compute(self, operands, operators):
    left, right = operands.pop(), operands.pop()
    op = operators.pop()
    if op == '+':
        operands.append(left + right)
    elif op == '-':
        operands.append(left - right)
    elif op == '*':
        operands.append(left * right)
    elif op == '/':
        operands.append(left / right)
```

populating-next-right-pointers-in-each-node-ii.py

```python
# Given a binary tree
#
# struct Node {
#   int val;
#   Node *left;
#   Node *right;
#   Node *next;
# }
#
#
# 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.
#
#
#
# Follow up:
#
#
#       You may only use constant extra space.
#       Recursive approach is fine, you may assume implicit stack space does not
# count as extra space for this problem.
#
#
#
# Example 1:
#
#
#
# Input: root = [1,2,3,4,5,null,7]
# Output: [1,#,2,3,#,4,5,7,#]
# Explanation: Given the above binary tree (Figure A), your function should
# populate each next pointer to point to its next right node, just like in Figure
# B. The serialized output is in level order as connected by the next pointers,
# with '#' signifying the end of each level.
#
#
#
# Constraints:
#
#
#       The number of nodes in the given tree is less than 6000.
#       -100 <= node.val <= 100# Time:  O(n)
# Space: O(1)

# Definition for a Node.
class Node(object):
    def __init__(self, val=0, left=None, right=None, next=None):
        self.val = val
        self.left = left
        self.right = right
        self.next = next


class Solution(object):
    # @param root, a tree node
    # @return nothing
```

```python
def connect(self, root):
    head = root
    pre = Node(0)
    cur = pre
    while root:
        while root:
            if root.left:
                cur.next = root.left
                cur = cur.next
            if root.right:
                cur.next = root.right
                cur = cur.next
            root = root.next
        root, cur = pre.next, pre
        cur.next = None
    return head
```

all-elements-in-two-binary-search-trees.py

```python
# Given two binary search trees root1 and root2.
#
# Return a list containing all the integers from both trees sorted in ascending
# order.
#
#
# Example 1:
#
# Input: root1 = [2,1,4], root2 = [1,0,3]
# Output: [0,1,1,2,3,4]
#
#
# Example 2:
#
# Input: root1 = [0,-10,10], root2 = [5,1,7,0,2]
# Output: [-10,0,0,1,2,5,7,10]
#
#
# Example 3:
#
# Input: root1 = [], root2 = [5,1,7,0,2]
# Output: [0,1,2,5,7]
#
#
# Example 4:
#
# Input: root1 = [0,-10,10], root2 = []
# Output: [-10,0,10]
#
#
# Example 5:
#
# Input: root1 = [1,null,8], root2 = [8,1]
# Output: [1,1,8,8]
#
#
#
# Constraints:
#
#
#       Each tree has at most 5000 nodes.
#       Each node's value is between [-10^5, 10^5].# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def getAllElements(self, root1, root2):
        """
        :type root1: TreeNode
        :type root2: TreeNode
        :rtype: List[int]
```

```python
"""
def inorder_gen(root):
    result, stack = [], [(root, False)]
    while stack:
        root, is_visited = stack.pop()
        if root is None:
            continue
        if is_visited:
            yield root.val
        else:
            stack.append((root.right, False))
            stack.append((root, True))
            stack.append((root.left, False))
    yield None

result = []
left_gen, right_gen = inorder_gen(root1), inorder_gen(root2)
left, right = next(left_gen), next(right_gen)
while left is not None or right is not None:
    if right is None or (left is not None and left < right):
        result.append(left)
        left = next(left_gen)
    else:
        result.append(right)
        right = next(right_gen)
return result
```

search-a-2d-matrix.py

```python
# 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.
#
#
# Example 1:
#
# Input:
# matrix = [
#   [1,   3,  5,  7],
#   [10, 11, 16, 20],
#   [23, 30, 34, 50]
# ]
# target = 3
# Output: true
#
#
# Example 2:
#
# Input:
# matrix = [
#   [1,   3,  5,  7],
#   [10, 11, 16, 20],
#   [23, 30, 34, 50]
# ]
# target = 13
# Output: false# Time:  O(logm + logn)
# Space: O(1)

class Solution(object):
    def searchMatrix(self, matrix, target):
        """
        :type matrix: List[List[int]]
        :type target: int
        :rtype: bool
        """
        if not matrix:
            return False

        m, n = len(matrix), len(matrix[0])
        left, right = 0, m * n
        while left < right:
            mid = left + (right - left) / 2
            if matrix[mid / n][mid % n] >= target:
                right = mid
            else:
                left = mid + 1

        return left < m * n and matrix[left / n][left % n] == target
```

find-all-anagrams-in-a-string.py

```python
# Given a string s and a non-empty string p, find all the start indices of p's
# anagrams in s.
#
# Strings consists of lowercase English letters only and the length of both
# strings s and p will not be larger than 20,100.
#
# The order of output does not matter.
#
# Example 1:
# Input:
# s: "cbaebabacd" p: "abc"
#
# Output:
# [0, 6]
#
# Explanation:
# The substring with start index = 0 is "cba", which is an anagram of "abc".
# The substring with start index = 6 is "bac", which is an anagram of "abc".
#
#
#
# Example 2:
# Input:
# s: "abab" p: "ab"
#
# Output:
# [0, 1, 2]
#
# Explanation:
# The substring with start index = 0 is "ab", which is an anagram of "ab".
# The substring with start index = 1 is "ba", which is an anagram of "ab".
# The substring with start index = 2 is "ab", which is an anagram of "ab".# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findAnagrams(self, s, p):
        """
        :type s: str
        :type p: str
        :rtype: List[int]
        """
        result = []

        cnts = [0] * 26
        for c in p:
            cnts[ord(c) - ord('a')] += 1

        left, right = 0, 0
        while right < len(s):
            cnts[ord(s[right]) - ord('a')] -= 1
            while left <= right and cnts[ord(s[right]) - ord('a')] < 0:
                cnts[ord(s[left]) - ord('a')] += 1
                left += 1
            if right - left + 1 == len(p):
                result.append(left)
            right += 1

        return result
```

fizz-buzz-multithreaded.py

```python
# Write a program that outputs the string representation of numbers from 1 to n,
# however:
#
#
#       If the number is divisible by 3, output "fizz".
#       If the number is divisible by 5, output "buzz".
#       If the number is divisible by both 3 and 5, output "fizzbuzz".
#
#
# For example, for n = 15, we output: 1, 2, fizz, 4, buzz, fizz, 7, 8, fizz,
# buzz, 11, fizz, 13, 14, fizzbuzz.
#
# Suppose you are given the following code:
#
# class FizzBuzz {
#   public FizzBuzz(int n) { ... }               // constructor
#   public void fizz(printFizz) { ... }          // only output "fizz"
#   public void buzz(printBuzz) { ... }          // only output "buzz"
#   public void fizzbuzz(printFizzBuzz) { ... }  // only output "fizzbuzz"
#   public void number(printNumber) { ... }      // only output the numbers
# }
#
# Implement a multithreaded version of FizzBuzz with four threads. The same
# instance of FizzBuzz will be passed to four different threads:
#
#
#       Thread A will call fizz() to check for divisibility of 3 and
# outputs fizz.
#       Thread B will call buzz() to check for divisibility of 5 and
# outputs buzz.
#       Thread C will call fizzbuzz() to check for divisibility of 3 and 5 and
# outputs fizzbuzz.
#       Thread D will call number() which should only output the numbers.# Time:  O(n)
# Space: O(1)

import threading


class FizzBuzz(object):
    def __init__(self, n):
        self.__n = n
        self.__curr = 0
        self.__cv = threading.Condition()

    # printFizz() outputs "fizz"
    def fizz(self, printFizz):
        """
        :type printFizz: method
        :rtype: void
        """
        for i in xrange(1, self.__n+1):
            with self.__cv:
                while self.__curr % 4 != 0:
                    self.__cv.wait()
                self.__curr += 1
                if i % 3 == 0 and i % 5 != 0:
                    printFizz()
                self.__cv.notify_all()
```

```python
    # printBuzz() outputs "buzz"
    def buzz(self, printBuzz):
        """
        :type printBuzz: method
        :rtype: void
        """
        for i in xrange(1, self.__n+1):
            with self.__cv:
                while self.__curr % 4 != 1:
                    self.__cv.wait()
                self.__curr += 1
                if i % 3 != 0 and i % 5 == 0:
                    printBuzz()
                self.__cv.notify_all()

    # printFizzBuzz() outputs "fizzbuzz"
    def fizzbuzz(self, printFizzBuzz):
        """
        :type printFizzBuzz: method
        :rtype: void
        """
        for i in xrange(1, self.__n+1):
            with self.__cv:
                while self.__curr % 4 != 2:
                    self.__cv.wait()
                self.__curr += 1
                if i % 3 == 0 and i % 5 == 0:
                    printFizzBuzz()
                self.__cv.notify_all()

    # printNumber(x) outputs "x", where x is an integer.
    def number(self, printNumber):
        """
        :type printNumber: method
        :rtype: void
        """
        for i in xrange(1, self.__n+1):
            with self.__cv:
                while self.__curr % 4 != 3:
                    self.__cv.wait()
                self.__curr += 1
                if i % 3 != 0 and i % 5 != 0:
                    printNumber(i)
                self.__cv.notify_all()
```

minimum-genetic-mutation.py

```
# A gene string can be represented by an 8-character long string, with choices
# from "A", "C", "G", "T".
#
# Suppose we need to investigate about a mutation (mutation from "start" to
# "end"), where ONE mutation is defined as ONE single character changed in the
# gene string.
#
# For example, "AACCGGTT" -> "AACCGGTA" is 1 mutation.
#
# Also, there is a given gene "bank", which records all the valid gene
# mutations. A gene must be in the bank to make it a valid gene string.
#
# Now, given 3 things - start, end, bank, your task is to determine what is the
# minimum number of mutations needed to mutate from "start" to "end". If there is
# no such a mutation, return -1.
#
# Note:
#
#
#       Starting point is assumed to be valid, so it might not be included in
# the bank.
#       If multiple mutations are needed, all mutations during in the sequence
# must be valid.
#       You may assume start and end string is not the same.
#
#
#
#
# Example 1:
#
# start: "AACCGGTT"
# end:   "AACCGGTA"
# bank: ["AACCGGTA"]
#
# return: 1
#
#
#
#
# Example 2:
#
# start: "AACCGGTT"
# end:   "AAACGGTA"
# bank: ["AACCGGTA", "AACCGCTA", "AAACGGTA"]
#
# return: 2
#
#
#
#
# Example 3:
#
# start: "AAAAACCC"
# end:   "AACCCCCC"
# bank: ["AAAACCCC", "AAACCCCC", "AACCCCCC"]
#
# return: 3# Time:  O(n * b), n is the length of gene string, b is size of bank
# Space: O(b)
```

```python
from collections import deque

class Solution(object):
    def minMutation(self, start, end, bank):
        """
        :type start: str
        :type end: str
        :type bank: List[str]
        :rtype: int
        """
        lookup = {}
        for b in bank:
            lookup[b] = False

        q = deque([(start, 0)])
        while q:
            cur, level = q.popleft()
            if cur == end:
                return level

            for i in xrange(len(cur)):
                for c in ['A', 'T', 'C', 'G']:
                    if cur[i] == c:
                        continue

                    next_str = cur[:i] + c + cur[i+1:]
                    if next_str in lookup and lookup[next_str] == False:
                        q.append((next_str, level+1))
                        lookup[next_str] = True

        return -1
```

alphabet-board-path.py

```python
# On an alphabet board, we start at position (0, 0), corresponding to
# character board[0][0].
#
# Here, board = ["abcde", "fghij", "klmno", "pqrst", "uvwxy", "z"], as shown in
# the diagram below.
#
#
#
# We may make the following moves:
#
#
#       'U' moves our position up one row, if the position exists on the board;
#       'D' moves our position down one row, if the position exists on the
# board;
#       'L' moves our position left one column, if the position exists on the
# board;
#       'R' moves our position right one column, if the position exists on the
# board;
#       '!' adds the character board[r][c] at our current position (r, c) to
# the answer.
#
#
# (Here, the only positions that exist on the board are positions with letters
# on them.)
#
# Return a sequence of moves that makes our answer equal to target in the
# minimum number of moves.  You may return any path that does so.
#
#
# Example 1:
# Input: target = "leet"
# Output: "DDR!UURRR!!DDD!"
# Example 2:
# Input: target = "code"
# Output: "RR!DDRR!UUL!R!"
#
#
# Constraints:
#
#
#       1 <= target.length <= 100
#       target consists only of English lowercase letters.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def alphabetBoardPath(self, target):
        """
        :type target: str
        :rtype: str
        """
        x, y = 0, 0
        result = []
        for c in target:
            y1, x1 = divmod(ord(c)-ord('a'), 5)
            result.append('U' * max(y-y1, 0))
            result.append('L' * max(x-x1, 0))
            result.append('R' * max(x1-x, 0))
            result.append('D' * max(y1-y, 0))
```

```
        result.append('!')
        x, y = x1, y1
    return "".join(result)
```

pancake-sorting.py

```python
# Given an array of integers A, We need to sort the array performing a series of
# pancake flips.
#
# In one pancake flip we do the following steps:
#
#
#       Choose an integer k where 0 <= k < A.length.
#       Reverse the sub-array A[0...k].
#
#
# For example, if A = [3,2,1,4] and we performed a pancake flip choosing k = 2,
# we reverse the sub-array [3,2,1], so A = [1,2,3,4] after the pancake flip at k =
# 2.
#
# Return an array of the k-values of the pancake flips that should be performed
# in order to sort A. Any valid answer that sorts the array within 10 * A.length
# flips will be judged as correct.
#
#
# Example 1:
#
# Input: A = [3,2,4,1]
# Output: [4,2,4,3]
# Explanation:
# We perform 4 pancake flips, with k values 4, 2, 4, and 3.
# Starting state: A = [3, 2, 4, 1]
# After 1st flip (k = 4): A = [1, 4, 2, 3]
# After 2nd flip (k = 2): A = [4, 1, 2, 3]
# After 3rd flip (k = 4): A = [3, 2, 1, 4]
# After 4th flip (k = 3): A = [1, 2, 3, 4], which is sorted.
# Notice that we return an array of the chosen k values of the pancake flips.
#
#
# Example 2:
#
# Input: A = [1,2,3]
# Output: []
# Explanation: The input is already sorted, so there is no need to flip
# anything.
# Note that other answers, such as [3, 3], would also be accepted.
#
#
#
# Constraints:
#
#
#       1 <= A.length <= 100
#       1 <= A[i] <= A.length
#       All integers in A are unique (i.e. A is a permutation of the integers
# from 1 to A.length).# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    def pancakeSort(self, A):
        """
        :type A: List[int]
        :rtype: List[int]
        """
```

```
def reverse(l, begin, end):
    for i in xrange((end-begin) // 2):
        l[begin+i], l[end-1-i] = l[end-1-i], l[begin+i]

result = []
for n in reversed(xrange(1, len(A)+1)):
    i = A.index(n)
    reverse(A, 0, i+1)
    result.append(i+1)
    reverse(A, 0, n)
    result.append(n)
return result
```

01-matrix.py

```python
# Given a matrix consists of 0 and 1, find the distance of the nearest 0 for
# each cell.
#
# The distance between two adjacent cells is 1.
#
#
#
# Example 1:
#
# Input:
# [[0,0,0],
#  [0,1,0],
#  [0,0,0]]
#
# Output:
# [[0,0,0],
#  [0,1,0],
#  [0,0,0]]
#
#
# Example 2:
#
# Input:
# [[0,0,0],
#  [0,1,0],
#  [1,1,1]]
#
# Output:
# [[0,0,0],
#  [0,1,0],
#  [1,2,1]]
#
#
#
#
# Note:
#
#
#       The number of elements of the given matrix will not exceed 10,000.
#       There are at least one 0 in the given matrix.
#       The cells are adjacent in only four directions: up, down, left and
# right.# Time:  O(m * n)
# Space: O(m * n)

import collections


class Solution(object):
    def updateMatrix(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: List[List[int]]
        """
        queue = collections.deque()
        for i in xrange(len(matrix)):
            for j in xrange(len(matrix[0])):
                if matrix[i][j] == 0:
                    queue.append((i, j))
```

```python
            else:
                matrix[i][j] = float("inf")

        dirs = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        while queue:
            cell = queue.popleft()
            for dir in dirs:
                i, j = cell[0]+dir[0], cell[1]+dir[1]
                if not (0 <= i < len(matrix)) or not (0 <= j < len(matrix[0])) or \
                   matrix[i][j] <= matrix[cell[0]][cell[1]]+1:
                        continue
                queue.append((i, j))
                matrix[i][j] = matrix[cell[0]][cell[1]]+1

        return matrix


# Time:  O(m * n)
# Space: O(m * n)
# dp solution
class Solution2(object):
    def updateMatrix(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: List[List[int]]
        """
        dp = [[float("inf")]*len(matrix[0]) for _ in xrange(len(matrix))]
        for i in xrange(len(matrix)):
            for j in xrange(len(matrix[i])):
                if matrix[i][j] == 0:
                    dp[i][j] = 0
                else:
                    if i > 0:
                        dp[i][j] = min(dp[i][j], dp[i-1][j]+1)
                    if j > 0:
                        dp[i][j] = min(dp[i][j], dp[i][j-1]+1)
        for i in reversed(xrange(len(matrix))):
            for j in reversed(xrange(len(matrix[i]))):
                if matrix[i][j] == 0:
                    dp[i][j] = 0
                else:
                    if i < len(matrix)-1:
                        dp[i][j] = min(dp[i][j], dp[i+1][j]+1)
                    if j < len(matrix[i])-1:
                        dp[i][j] = min(dp[i][j], dp[i][j+1]+1)
        return dp
```

maximum-product-subarray.py

```python
# Given an integer array nums, find the contiguous subarray within an array
# (containing at least one number) which has the largest product.
#
# Example 1:
#
# Input: [2,3,-2,4]
# Output: 6
# Explanation: [2,3] has the largest product 6.
#
#
# Example 2:
#
# Input: [-2,0,-1]
# Output: 0
# Explanation: The result cannot be 2, because [-2,-1] is not a subarray.# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @param A, a list of integers
    # @return an integer
    def maxProduct(self, A):
        global_max, local_max, local_min = float("-inf"), 1, 1
        for x in A:
            local_max, local_min = max(x, local_max * x, local_min * x), min(x, local_max * x, local_min * x)
            global_max = max(global_max, local_max)
        return global_max

class Solution2(object):
    # @param A, a list of integers
    # @return an integer
    def maxProduct(self, A):
        global_max, local_max, local_min = float("-inf"), 1, 1
        for x in A:
            local_max = max(1, local_max)
            if x > 0:
                local_max, local_min = local_max * x, local_min * x
            else:
                local_max, local_min = local_min * x, local_max * x
            global_max = max(global_max, local_max)
        return global_max
```

distant-barcodes.py

```python
# Example 2:
#
# Input: [1,1,1,1,2,2,3,3]
# Output: [1,3,1,3,2,1,2,1]# Time:  O(klogk), k is the number of distinct barcodes
# Space: O(k)

import collections


class Solution(object):
    def rearrangeBarcodes(self, barcodes):
        """
        :type barcodes: List[int]
        :rtype: List[int]
        """
        cnts = collections.Counter(barcodes)
        sorted_cnts = [[v, k] for k, v in cnts.iteritems()]
        sorted_cnts.sort(reverse=True)

        i = 0
        for v, k in sorted_cnts:
            for _ in xrange(v):
                barcodes[i] = k
                i += 2
                if i >= len(barcodes):
                    i = 1
        return barcodes
```

binary-subarrays-with-sum.py

```python
# In an array A of 0s and 1s, how many non-empty subarrays have sum S?
#
#
#
# Example 1:
#
# Input: A = [1,0,1,0,1], S = 2
# Output: 4
# Explanation:
# The 4 subarrays are bolded below:
# [1,0,1,0,1]
# [1,0,1,0,1]
# [1,0,1,0,1]
# [1,0,1,0,1]
#
#
#
#
# Note:
#
#
#       A.length <= 30000
#       0 <= S <= A.length
#       A[i] is either 0 or 1.# Time:  O(n)
# Space: O(1)

# Two pointers solution
class Solution(object):
    def numSubarraysWithSum(self, A, S):
        """
        :type A: List[int]
        :type S: int
        :rtype: int
        """
        result = 0
        left, right, sum_left, sum_right = 0, 0, 0, 0
        for i, a in enumerate(A):
            sum_left += a
            while left < i and sum_left > S:
                sum_left -= A[left]
                left += 1
            sum_right += a
            while right < i and \
                  (sum_right > S or (sum_right == S and not A[right])):
                sum_right -= A[right]
                right += 1
            if sum_left == S:
                result += right-left+1
        return result
```

delete-node-in-a-bst.py

```
# Given a root node reference of a BST and a key, delete the node with the given
# key in the BST. Return the root node reference (possibly updated) of the BST.
#
# Basically, the deletion can be divided into two stages:
#
# Search for a node to remove.
# If the node is found, delete the node.
#
#
#
# Note: Time complexity should be O(height of tree).
#
# Example:
# root = [5,3,6,2,4,null,7]
# key = 3
#
#     5
#    / \
#   3   6
#  / \   \
# 2   4   7
#
# Given key to delete is 3. So we find the node with value 3 and delete it.
#
# One valid answer is [5,4,6,2,null,null,7], shown in the following BST.
#
#     5
#    / \
#   4   6
#  /     \
# 2       7
#
# Another valid answer is [5,2,6,null,4,null,7].
#
#     5
#    / \
#   2   6
#    \   \
#     4   7# Time:  O(h)
# Space: O(h)

class Solution(object):
    def deleteNode(self, root, key):
        """
        :type root: TreeNode
        :type key: int
        :rtype: TreeNode
        """
        if not root:
            return root

        if root.val > key:
            root.left = self.deleteNode(root.left, key)
        elif root.val < key:
            root.right = self.deleteNode(root.right, key)
        else:
            if not root.left:
                right = root.right
```

```python
            del root
            return right
        elif not root.right:
            left = root.left
            del root
            return left
        else:
            successor = root.right
            while successor.left:
                successor = successor.left

            root.val = successor.val
            root.right = self.deleteNode(root.right, successor.val)

    return root
```

random-pick-index.py

```python
# Given an array of integers with possible duplicates, randomly output the index
# of a given target number. You can assume that the given target number must exist
# in the array.
#
# Note:
#
# The array size can be very large. Solution that uses too much extra space will
# not pass the judge.
#
# Example:
#
# int[] nums = new int[] {1,2,3,3,3};
# Solution solution = new Solution(nums);
#
# // pick(3) should return either index 2, 3, or 4 randomly. Each index should
# have equal probability of returning.
# solution.pick(3);
#
# // pick(1) should return 0. Since in the array only nums[0] is equal to 1.
# solution.pick(1);# Time:  O(n)
# Space: O(1)

from random import randint

class Solution(object):

    def __init__(self, nums):
        """

        :type nums: List[int]
        :type numsSize: int
        """
        self.__nums = nums

    def pick(self, target):
        """
        :type target: int
        :rtype: int
        """
        reservoir = -1
        n = 0
        for i in xrange(len(self.__nums)):
            if self.__nums[i] != target:
                continue
            reservoir = i if randint(1, n+1) == 1 else reservoir
            n += 1
        return reservoir
```

word-break.py

```
# Given a non-empty string s and a dictionary wordDict containing a list of non-
# empty words, determine if s can be segmented into a space-separated sequence of
# one or more dictionary words.
#
# Note:
#
#
#       The same word in the dictionary may be reused multiple times in the
# segmentation.
#       You may assume the dictionary does not contain duplicate words.
#
#
# Example 1:
#
# Input: s = "leetcode", wordDict = ["leet", "code"]
# Output: true
# Explanation: Return true because "leetcode" can be segmented as "leet code".
#
#
# Example 2:
#
# Input: s = "applepenapple", wordDict = ["apple", "pen"]
# Output: true
# Explanation: Return true because "applepenapple" can be segmented as "apple
# pen apple".
#              Note that you are allowed to reuse a dictionary word.
#
#
# Example 3:
#
# Input: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
# Output: false# Time:  O(n * l^2)
# Space: O(n)

class Solution(object):
    def wordBreak(self, s, wordDict):
        """
        :type s: str
        :type wordDict: Set[str]
        :rtype: bool
        """
        n = len(s)

        max_len = 0
        for string in wordDict:
            max_len = max(max_len, len(string))

        can_break = [False for _ in xrange(n + 1)]
        can_break[0] = True
        for i in xrange(1, n + 1):
            for l in xrange(1, min(i, max_len) + 1):
                if can_break[i-l] and s[i-l:i] in wordDict:
                    can_break[i] = True
                    break

        return can_break[-1]
```

video-stitching.py

```
# You are given a series of video clips from a sporting event that lasted T
# seconds.  These video clips can be overlapping with each other and have varied
# lengths.
#
# Each video clip clips[i] is an interval: it starts at time clips[i][0] and
# ends at time clips[i][1].  We can cut these clips into segments freely: for
# example, a clip [0, 7] can be cut into segments [0, 1] + [1, 3] + [3, 7].
#
# Return the minimum number of clips needed so that we can cut the clips into
# segments that cover the entire sporting event ([0, T]).  If the task is
# impossible, return -1.
#
#
#
# Example 1:
#
# Input: clips = [[0,2],[4,6],[8,10],[1,9],[1,5],[5,9]], T = 10
# Output: 3
# Explanation:
# We take the clips [0,2], [8,10], [1,9]; a total of 3 clips.
# Then, we can reconstruct the sporting event as follows:
# We cut [1,9] into segments [1,2] + [2,8] + [8,9].
# Now we have segments [0,2] + [2,8] + [8,10] which cover the sporting event [0,
# 10].
#
#
# Example 2:
#
# Input: clips = [[0,1],[1,2]], T = 5
# Output: -1
# Explanation:
# We can't cover [0,5] with only [0,1] and [1,2].
#
#
# Example 3:
#
# Input: clips = [[0,1],[6,8],[0,2],[5,6],[0,4],[0,3],[6,7],[1,3],[4,7],[1,4],[2
# ,5],[2,6],[3,4],[4,5],[5,7],[6,9]], T = 9
# Output: 3
# Explanation:
# We can take clips [0,4], [4,7], and [6,9].
#
#
# Example 4:
#
# Input: clips = [[0,4],[2,8]], T = 5
# Output: 2
# Explanation:
# Notice you can have extra video after the event ends.
#
#
#
# Constraints:
#
#
#       1 <= clips.length <= 100
#       0 <= clips[i][0] <= clips[i][1] <= 100
#       0 <= T <= 100# Time:  O(nlogn)
```

```python
# Space: O(1)

class Solution(object):
    def videoStitching(self, clips, T):
        """
        :type clips: List[List[int]]
        :type T: int
        :rtype: int
        """
        result = 1
        curr_reachable, reachable = 0, 0
        clips.sort()
        for left, right in clips:
            if left > reachable:
                break
            elif left > curr_reachable:
                curr_reachable = reachable
                result += 1
            reachable = max(reachable, right)
            if reachable >= T:
                return result
        return -1
```

shortest-path-with-alternating-colors.py

```
# Consider a directed graph, with nodes labelled 0, 1, ..., n-1.  In this graph,
# each edge is either red or blue, and there could be self-edges or parallel
# edges.
#
# Each [i, j] in red_edges denotes a red directed edge from node i to node j.
# Similarly, each [i, j] in blue_edges denotes a blue directed edge from node i to
# node j.
#
# Return an array answer of length n, where each answer[X] is the length of the
# shortest path from node 0 to node X such that the edge colors alternate along
# the path (or -1 if such a path doesn't exist).
#
#
# Example 1:
# Input: n = 3, red_edges = [[0,1],[1,2]], blue_edges = []
# Output: [0,1,-1]
# Example 2:
# Input: n = 3, red_edges = [[0,1]], blue_edges = [[2,1]]
# Output: [0,1,-1]
# Example 3:
# Input: n = 3, red_edges = [[1,0]], blue_edges = [[2,1]]
# Output: [0,-1,-1]
# Example 4:
# Input: n = 3, red_edges = [[0,1]], blue_edges = [[1,2]]
# Output: [0,1,2]
# Example 5:
# Input: n = 3, red_edges = [[0,1],[0,2]], blue_edges = [[1,0]]
# Output: [0,1,1]
#
#
# Constraints:
#
#
#       1 <= n <= 100
#       red_edges.length <= 400
#       blue_edges.length <= 400
#       red_edges[i].length == blue_edges[i].length == 2
#       0 <= red_edges[i][j], blue_edges[i][j] < n# Time:  O(n + e), e is the number of red and blue edges
# Space: O(n + e)

import collections


class Solution(object):
    def shortestAlternatingPaths(self, n, red_edges, blue_edges):
        """
        :type n: int
        :type red_edges: List[List[int]]
        :type blue_edges: List[List[int]]
        :rtype: List[int]
        """
        neighbors = [[set() for _ in xrange(2)] for _ in xrange(n)]
        for i, j in red_edges:
            neighbors[i][0].add(j)
        for i, j in blue_edges:
            neighbors[i][1].add(j)
        INF = max(2*n-3, 0)+1
        dist = [[INF, INF] for i in xrange(n)]
```

```python
dist[0] = [0, 0]
q = collections.deque([(0, 0), (0, 1)])
while q:
    i, c = q.popleft()
    for j in neighbors[i][c]:
        if dist[j][c] != INF:
            continue
        dist[j][c] = dist[i][1^c]+1
        q.append((j, 1^c))
return [x if x != INF else -1 for x in map(min, dist)]
```

maximum-width-ramp.py

```python
# Note:
#
#
#       2 <= A.length <= 50000
#       0 <= A[i] <= 50000# Time:  O(n)
# Space: O(n)

class Solution(object):
    def maxWidthRamp(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        result = 0
        s = []
        for i in A:
            if not s or A[s[-1]] > A[i]:
                s.append(i)
        for j in reversed(xrange(len(A))):
            while s and A[s[-1]] <= A[j]:
                result = max(result, j-s.pop())
        return result
```

binary-search-tree-to-greater-sum-tree.py

```python
# Note: This question is the same as 538: https://leetcode.com/problems/convert-
# bst-to-greater-tree/# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def bstToGst(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        def bstToGstHelper(root, prev):
            if not root:
                return root
            bstToGstHelper(root.right, prev)
            root.val += prev[0]
            prev[0] = root.val
            bstToGstHelper(root.left, prev)
            return root

        prev = [0]
        return bstToGstHelper(root, prev)
```

nth-digit.py

```python
# Find the nth digit of the infinite integer sequence 1, 2, 3, 4, 5, 6, 7, 8, 9,
# 10, 11, ...
#
# Note:
#
# n is positive and will fit within the range of a 32-bit signed integer (n <
# 231).
#
#
# Example 1:
# Input:
# 3
#
# Output:
# 3
#
#
#
# Example 2:
# Input:
# 11
#
# Output:
# 0
#
# Explanation:
# The 11th digit of the sequence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ... is a 0,
# which is part of the number 10.# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def findNthDigit(self, n):
        """
        :type n: int
        :rtype: int
        """
        digit_len = 1
        while n > digit_len * 9 * (10 ** (digit_len-1)):
            n -= digit_len  * 9 * (10 ** (digit_len-1))
            digit_len += 1

        num = 10 ** (digit_len-1) + (n-1)/digit_len

        nth_digit = num / (10 ** ((digit_len-1) - ((n-1)%digit_len)))
        nth_digit %= 10

        return nth_digit
```

elimination-game.py

```python
# There is a list of sorted integers from 1 to n. Starting from left to right,
# remove the first number and every other number afterward until you reach the end
# of the list.
#
# Repeat the previous step again, but this time from right to left, remove the
# right most number and every other number from the remaining numbers.
#
# We keep repeating the steps again, alternating left to right and right to
# left, until a single number remains.
#
# Find the last number that remains starting with a list of length n.
#
# Example:
# Input:
# n = 9,
# 1 2 3 4 5 6 7 8 9
# 2 4 6 8
# 2 6
# 6
#
# Output:
# 6# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def lastRemaining(self, n):
        """
        :type n: int
        :rtype: int
        """
        start, step, direction = 1, 2, 1
        while n > 1:
            start += direction * (step * (n/2) - step/2)
            n /= 2
            step *= 2
            direction *= -1
        return start
```

can-make-palindrome-from-substring.py

```python
# Given a string s, we make queries on substrings of s.
#
# For each query queries[i] = [left, right, k], we may rearrange the substring
# s[left], ..., s[right], and then choose up to k of them to replace with any
# lowercase English letter.
#
# If the substring is possible to be a palindrome string after the operations
# above, the result of the query is true. Otherwise, the result is false.
#
# Return an array answer[], where answer[i] is the result of the i-th query
# queries[i].
#
# Note that: Each letter is counted individually for replacement so if for
# example s[left..right] = "aaa", and k = 2, we can only replace two of the
# letters.  (Also, note that the initial string s is never modified by any query.)
#
#
# Example :
#
# Input: s = "abcda", queries = [[3,3,0],[1,2,0],[0,3,1],[0,3,2],[0,4,1]]
# Output: [true,false,false,true,true]
# Explanation:
# queries[0] : substring = "d", is palidrome.
# queries[1] : substring = "bc", is not palidrome.
# queries[2] : substring = "abcd", is not palidrome after replacing only 1
# character.
# queries[3] : substring = "abcd", could be changed to "abba" which is
# palidrome. Also this can be changed to "baab" first rearrange it "bacd" then
# replace "cd" with "ab".
# queries[4] : substring = "abcda", could be changed to "abcba" which is
# palidrome.
#
#
#
# Constraints:
#
#
#       1 <= s.length, queries.length <= 10^5
#       0 <= queries[i][0] <= queries[i][1] < s.length
#       0 <= queries[i][2] <= s.length
#       s only contains lowercase English letters.# Time:  O(m + n), m is the number of queries, n is the leng
# Space: O(n)

import itertools


class Solution(object):
    def canMakePaliQueries(self, s, queries):
        """
        :type s: str
        :type queries: List[List[int]]
        :rtype: List[bool]
        """
        CHARSET_SIZE = 26
        curr, count = [0]*CHARSET_SIZE, [[0]*CHARSET_SIZE]
        for c in s:
            curr[ord(c)-ord('a')] += 1
            count.append(curr[:])
```

```python
return [sum((b-a)%2 for a, b in itertools.izip(count[left], count[right+1]))//2 <= k
        for left, right, k in queries]
```

design-twitter.py

```python
# Design a simplified version of Twitter where users can post tweets,
# follow/unfollow another user and is able to see the 10 most recent tweets in the
# user's news feed. Your design should support the following methods:
#
#
#
# postTweet(userId, tweetId): Compose a new tweet.
# getNewsFeed(userId): Retrieve the 10 most recent tweet ids in the user's news
# feed. Each item in the news feed must be posted by users who the user followed
# or by the user herself. Tweets must be ordered from most recent to least recent.
# follow(followerId, followeeId): Follower follows a followee.
# unfollow(followerId, followeeId): Follower unfollows a followee.
#
#
#
# Example:
# Twitter twitter = new Twitter();
#
# // User 1 posts a new tweet (id = 5).
# twitter.postTweet(1, 5);
#
# // User 1's news feed should return a list with 1 tweet id -> [5].
# twitter.getNewsFeed(1);
#
# // User 1 follows user 2.
# twitter.follow(1, 2);
#
# // User 2 posts a new tweet (id = 6).
# twitter.postTweet(2, 6);
#
# // User 1's news feed should return a list with 2 tweet ids -> [6, 5].
# // Tweet id 6 should precede tweet id 5 because it is posted after tweet id 5.
# twitter.getNewsFeed(1);
#
# // User 1 unfollows user 2.
# twitter.unfollow(1, 2);
#
# // User 1's news feed should return a list with 1 tweet id -> [5],
# // since user 1 is no longer following user 2.
# twitter.getNewsFeed(1);# Time:  O(klogu), k is most recently number of tweets,
#                   u is the number of the user's following.
# Space: O(t + f), t is the total number of tweets,
#                   f is the total number of followings.

import collections
import heapq


class Twitter(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.__number_of_most_recent_tweets = 10
        self.__followings = collections.defaultdict(set)
        self.__messages = collections.defaultdict(list)
        self.__time = 0
```

```python
def postTweet(self, userId, tweetId):
    """
    Compose a new tweet.
    :type userId: int
    :type tweetId: int
    :rtype: void
    """
    self.__time += 1
    self.__messages[userId].append((self.__time, tweetId))

def getNewsFeed(self, userId):
    """
    Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be post
    :type userId: int
    :rtype: List[int]
    """
    max_heap = []
    if self.__messages[userId]:
        heapq.heappush(max_heap, (-self.__messages[userId][-1][0], userId, 0))
    for uid in self.__followings[userId]:
        if self.__messages[uid]:
            heapq.heappush(max_heap, (-self.__messages[uid][-1][0], uid, 0))

    result = []
    while max_heap and len(result) < self.__number_of_most_recent_tweets:
        t, uid, curr = heapq.heappop(max_heap)
        nxt = curr + 1
        if nxt != len(self.__messages[uid]):
            heapq.heappush(max_heap, (-self.__messages[uid][-(nxt+1)][0], uid, nxt))
        result.append(self.__messages[uid][-(curr+1)][1])
    return result

def follow(self, followerId, followeeId):
    """
    Follower follows a followee. If the operation is invalid, it should be a no-op.
    :type followerId: int
    :type followeeId: int
    :rtype: void
    """
    if followerId != followeeId:
        self.__followings[followerId].add(followeeId)

def unfollow(self, followerId, followeeId):
    """
    Follower unfollows a followee. If the operation is invalid, it should be a no-op.
    :type followerId: int
    :type followeeId: int
    :rtype: void
    """
    self.__followings[followerId].discard(followeeId)
```

palindrome-partitioning.py

```python
# Given a string s, partition s such that every substring of the partition is a
# palindrome.
#
# Return all possible palindrome partitioning of s.
#
# Example:
#
# Input: "aab"
# Output:
# [
#   ["aa","b"],
#   ["a","a","b"]
# ]# Time:  O(n^2 ~ 2^n)
# Space: O(n^2)

class Solution(object):
    # @param s, a string
    # @return a list of lists of string
    def partition(self, s):
        n = len(s)

        is_palindrome = [[0 for j in xrange(n)] for i in xrange(n)]
        for i in reversed(xrange(0, n)):
            for j in xrange(i, n):
                is_palindrome[i][j] = s[i] == s[j] and ((j - i < 2 ) or is_palindrome[i + 1][j - 1])

        sub_partition = [[] for i in xrange(n)]
        for i in reversed(xrange(n)):
            for j in xrange(i, n):
                if is_palindrome[i][j]:
                    if j + 1 < n:
                        for p in sub_partition[j + 1]:
                            sub_partition[i].append([s[i:j + 1]] + p)
                    else:
                        sub_partition[i].append([s[i:j + 1]])

        return sub_partition[0]

# Time:  O(2^n)
# Space: O(n)
# recursive solution
class Solution2(object):
    # @param s, a string
    # @return a list of lists of string
    def partition(self, s):
        result = []
        self.partitionRecu(result, [], s, 0)
        return result

    def partitionRecu(self, result, cur, s, i):
        if i == len(s):
            result.append(list(cur))
        else:
            for j in xrange(i, len(s)):
                if self.isPalindrome(s[i: j + 1]):
                    cur.append(s[i: j + 1])
                    self.partitionRecu(result, cur, s, j + 1)
                    cur.pop()
```

```python
def isPalindrome(self, s):
    for i in xrange(len(s) / 2):
        if s[i] != s[-(i + 1)]:
            return False
    return True
```

rectangle-area.py

```python
# 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.
#
#
#
# Example:
#
# Input: A = -3, B = 0, C = 3, D = 4, E = 0, F = -1, G = 9, H = 2
# Output: 45
#
# Note:
#
# Assume that the total area is never beyond the maximum possible value of int.# Time:  O(1)
# Space: O(1)

class Solution(object):
    # @param {integer} A
    # @param {integer} B
    # @param {integer} C
    # @param {integer} D
    # @param {integer} E
    # @param {integer} F
    # @param {integer} G
    # @param {integer} H
    # @return {integer}
    def computeArea(self, A, B, C, D, E, F, G, H):
        return (D - B) * (C - A) + \
               (G - E) * (H - F) - \
               max(0, (min(C, G) - max(A, E))) * \
               max(0, (min(D, H) - max(B, F)))
```

course-schedule-ii.py

```python
# There are a total of n courses you have to take labelled from 0 to n - 1.
#
# Some courses may have prerequisites, for example, if prerequisites[i] = [ai,
# bi] this means you must take the course bi before the course ai.
#
# Given the total number of courses numCourses and a list of the prerequisite
# pairs, return the ordering of courses you should take to finish all courses.
#
# If there are many valid answers, return any of them. If it is impossible to
# finish all courses, return an empty array.
#
#
# Example 1:
#
# Input: numCourses = 2, prerequisites = [[1,0]]
# Output: [0,1]
# Explanation: 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].
#
#
# Example 2:
#
# Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
# Output: [0,2,1,3]
# Explanation: 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].
#
#
# Example 3:
#
# Input: numCourses = 1, prerequisites = []
# Output: [0]
#
#
#
# Constraints:
#
#
#       1 <= numCourses <= 2000
#       0 <= prerequisites.length <= numCourses * (numCourses - 1)
#       prerequisites[i].length == 2
#       0 <= ai, bi < numCourses
#       ai != bi
#       All the pairs [ai, bi] are distinct.from collections import defaultdict, deque


class Solution(object):
    def findOrder(self, numCourses, prerequisites):
        """
        :type numCourses: int
        :type prerequisites: List[List[int]]
        :rtype: List[int]
        """
        res, zero_in_degree_queue = [], deque()
        in_degree, out_degree = defaultdict(set), defaultdict(set)
```

```python
for i, j in prerequisites:
    in_degree[i].add(j)
    out_degree[j].add(i)

for i in xrange(numCourses):
    if i not in in_degree:
        zero_in_degree_queue.append(i)

while zero_in_degree_queue:
    prerequisite = zero_in_degree_queue.popleft()
    res.append(prerequisite)

    if prerequisite in out_degree:
        for course in out_degree[prerequisite]:
            in_degree[course].discard(prerequisite)
            if not in_degree[course]:
                zero_in_degree_queue.append(course)

        del out_degree[prerequisite]

if out_degree:
    return []

return res
```

car-pooling.py

```python
# Example 2:
#
# Input: trips = [[2,1,5],[3,3,7]], capacity = 5
# Output: true
#
#
#
# Example 3:
#
# Input: trips = [[2,1,5],[3,5,7]], capacity = 3
# Output: true
#
#
#
# Example 4:
#
# Input: trips = [[3,2,7],[3,7,9],[8,3,9]], capacity = 11
# Output: true# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def carPooling(self, trips, capacity):
        """
        :type trips: List[List[int]]
        :type capacity: int
        :rtype: bool
        """
        line = [x for num, start, end in trips for x in [[start, num], [end, -num]]]
        line.sort()
        for _, num in line:
            capacity -= num
            if capacity < 0:
                return False
        return True
```

maximum-nesting-depth-of-two-valid-parentheses-strings.py

```python
# A string is a valid parentheses string (denoted VPS) if and only if it
# consists of "(" and ")" characters only, and:
#
#
#       It is the empty string, or
#       It can be written as AB (A concatenated with B), where A and B are
# VPS's, or
#       It can be written as (A), where A is a VPS.
#
#
# We can similarly define the nesting depth depth(S) of any VPS S as follows:
#
#
#       depth("") = 0
#       depth(A + B) = max(depth(A), depth(B)), where A and B are VPS's
#       depth("(" + A + ")") = 1 + depth(A), where A is a VPS.
#
#
# For example,  "", "()()", and "()(()())" are VPS's (with nesting depths 0, 1,
# and 2), and ")(" and "(()" are not VPS's.
#
#
#
# Given a VPS seq, split it into two disjoint subsequences A and B, such that A
# and B are VPS's (and A.length + B.length = seq.length).
#
# Now choose any such A and B such that max(depth(A), depth(B)) is the minimum
# possible value.
#
# Return an answer array (of length seq.length) that encodes such a choice of A
# and B:  answer[i] = 0 if seq[i] is part of A, else answer[i] = 1.  Note that
# even though multiple answers may exist, you may return any of them.
#
#
# Example 1:
#
# Input: seq = "(()())"
# Output: [0,1,1,1,1,0]
#
#
# Example 2:
#
# Input: seq = "()(())()"
# Output: [0,0,0,1,1,0,1,1]
#
#
#
# Constraints:
#
#
#       1 <= seq.size <= 10000# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxDepthAfterSplit(self, seq):
        """
        :type seq: str
        :rtype: List[int]
```

```python
        """
        return [(i & 1) ^ (seq[i] == '(') for i, c in enumerate(seq)]


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def maxDepthAfterSplit(self, seq):
        """
        :type seq: str
        :rtype: List[int]
        """
        A, B = 0, 0
        result = [0]*len(seq)
        for i, c in enumerate(seq):
            point = 1 if c == '(' else -1
            if (point == 1 and A <= B) or \
               (point == -1 and A >= B):
                A += point
            else:
                B += point
                result[i] = 1
        return result
```

maximum-level-sum-of-a-binary-tree.py

```python
# Given the root of a binary tree, the level of its root is 1, the level of its
# children is 2, and so on.
#
# Return the smallest level X such that the sum of all the values of nodes at
# level X is maximal.
#
#
#
# Example 1:
#
#
#
# Input: [1,7,0,7,-8,null,null]
# Output: 2
# Explanation:
# Level 1 sum = 1.
# Level 2 sum = 7 + 0 = 7.
# Level 3 sum = 7 + -8 = -1.
# So we return the level with the maximum sum which is level 2.
#
#
#
#
# Note:
#
#
#       The number of nodes in the given tree is between 1 and 10^4.
#       -10^5 <= node.val <= 10^5# Time:  O(n)
# Space: O(h)

import collections


# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


# dfs solution
class Solution(object):
    def maxLevelSum(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def dfs(node, i, level_sums):
            if not node:
                return
            if i == len(level_sums):
                level_sums.append(0)
            level_sums[i] += node.val
            dfs(node.left, i+1, level_sums)
            dfs(node.right, i+1, level_sums)

        level_sums = []
```

```python
        dfs(root, 0, level_sums)
        return level_sums.index(max(level_sums))+1


# Time:  O(n)
# Space: O(w)
# bfs solution
class Solution2(object):
    def maxLevelSum(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        result, level, max_total = 0, 1, float("-inf")
        q = collections.deque([root])
        while q:
            total = 0
            for _ in xrange(len(q)):
                node = q.popleft()
                total += node.val
                if node.left:
                    q.append(node.left)
                if node.right:
                    q.append(node.right)
            if total > max_total:
                result, max_total = level, total
            level += 1
        return result
```

combination-sum.py

```
# Given a set of candidate numbers (candidates) (without duplicates) and a
# target number (target), find all unique combinations in candidates where the
# candidate numbers sums to target.
#
# The same repeated number may be chosen from candidates unlimited number of
# times.
#
# Note:
#
#
#       All numbers (including target) will be positive integers.
#       The solution set must not contain duplicate combinations.
#
#
# Example 1:
#
# Input: candidates = [2,3,6,7], target = 7,
# A solution set is:
# [
#   [7],
#   [2,2,3]
# ]
#
#
# Example 2:
#
# Input: candidates = [2,3,5], target = 8,
# A solution set is:
# [
#   [2,2,2,2],
#   [2,3,3],
#   [3,5]
# ]
#
#
#
# Constraints:
#
#
#       1 <= candidates.length <= 30
#       1 <= candidates[i] <= 200
#       Each element of candidate is unique.
#       1 <= target <= 500# Time:  O(k * n^k)
# Space: O(k)

class Solution(object):
    # @param candidates, a list of integers
    # @param target, integer
    # @return a list of lists of integers
    def combinationSum(self, candidates, target):
        result = []
        self.combinationSumRecu(sorted(candidates), result, 0, [], target)
        return result

    def combinationSumRecu(self, candidates, result, start, intermediate, target):
        if target == 0:
            result.append(list(intermediate))
        while start < len(candidates) and candidates[start] <= target:
```

```
intermediate.append(candidates[start])
self.combinationSumRecu(candidates, result, start, intermediate, target - candidates[start])
intermediate.pop()
start += 1
```

flip-string-to-monotone-increasing.py

```python
# A string of '0's and '1's is monotone increasing if it consists of some number
# of '0's (possibly 0), followed by some number of '1's (also possibly 0.)
#
# We are given a string S of '0's and '1's, and we may flip any '0' to a '1' or
# a '1' to a '0'.
#
# Return the minimum number of flips to make S monotone increasing.
#
#
#
#
# Example 1:
#
# Input: "00110"
# Output: 1
# Explanation: We flip the last digit to get 00111.
#
#
#
# Example 2:
#
# Input: "010110"
# Output: 2
# Explanation: We flip to get 011111, or alternatively 000111.
#
#
#
# Example 3:
#
# Input: "00011000"
# Output: 2
# Explanation: We flip to get 00000000.
#
#
#
#
# Note:
#
#
#       1 <= S.length <= 20000
#       S only consists of '0' and '1' characters.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def minFlipsMonoIncr(self, S):
        """
        :type S: str
        :rtype: int
        """
        flip0, flip1 = 0, 0
        for c in S:
            flip0 += int(c == '1')
            flip1 = min(flip0, flip1 + int(c == '0'))
        return flip1
```

grumpy-bookstore-owner.py

```python
# Today, the bookstore owner has a store open for customers.length minutes.
# Every minute, some number of customers (customers[i]) enter the store, and all
# those customers leave after the end of that minute.
#
# On some minutes, the bookstore owner is grumpy.  If the bookstore owner is
# grumpy on the i-th minute, grumpy[i] = 1, otherwise grumpy[i] = 0.  When the
# bookstore owner is grumpy, the customers of that minute are not satisfied,
# otherwise they are satisfied.
#
# The bookstore owner knows a secret technique to keep themselves not grumpy for
# X minutes straight, but can only use it once.
#
# Return the maximum number of customers that can be satisfied throughout the
# day.
#
#
#
# Example 1:
#
# Input: customers = [1,0,1,2,1,1,7,5], grumpy = [0,1,0,1,0,1,0,1], X = 3
# Output: 16
# Explanation: The bookstore owner keeps themselves not grumpy for the last 3
# minutes.
# The maximum number of customers that can be satisfied = 1 + 1 + 1 + 1 + 7 + 5
# = 16.
#
#
#
#
# Note:
#
#
#       1 <= X <= customers.length == grumpy.length <= 20000
#       0 <= customers[i] <= 1000
#       0 <= grumpy[i] <= 1# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxSatisfied(self, customers, grumpy, X):
        """
        :type customers: List[int]
        :type grumpy: List[int]
        :type X: int
        :rtype: int
        """
        result, max_extra, extra = 0, 0, 0
        for i in xrange(len(customers)):
            result += 0 if grumpy[i] else customers[i]
            extra += customers[i] if grumpy[i] else 0
            if i >= X:
                extra -= customers[i-X] if grumpy[i-X] else 0
            max_extra = max(max_extra, extra)
        return result + max_extra
```

lowest-common-ancestor-of-a-binary-tree.py

```python
# 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 p and q as the lowest node in T that has both p and
# q as descendants (where we allow a node to be a descendant of itself)."
#
# Given the following binary tree:  root = [3,5,1,6,2,0,8,null,null,7,4]
#
#
#
# Example 1:
#
# Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
# Output: 3
# Explanation: The LCA of nodes 5 and 1 is 3.
#
#
# Example 2:
#
# Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
# Output: 5
# Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant
# of itself according to the LCA definition.
#
#
#
#
# Note:
#
#
#       All of the nodes' values will be unique.
#       p and q are different and both values will exist in the binary tree.# Time:  O(n)
# Space: O(h)

class Solution(object):
    # @param {TreeNode} root
    # @param {TreeNode} p
    # @param {TreeNode} q
    # @return {TreeNode}
    def lowestCommonAncestor(self, root, p, q):
        if root in (None, p, q):
            return root

        left, right = [self.lowestCommonAncestor(child, p, q) \
                          for child in (root.left, root.right)]
        # 1. If the current subtree contains both p and q,
        #    return their LCA.
        # 2. If only one of them is in that subtree,
        #    return that one of them.
        # 3. If neither of them is in that subtree,
        #    return the node of that subtree.
        return root if left and right else left or right
```

spiral-matrix-iii.py

```python
# Note:
#
#
#       1 <= R <= 100
#       1 <= C <= 100
#       0 <= r0 < R
#       0 <= c0 < C# Time:  O(max(m, n)^2)
# Space: O(1)

class Solution(object):
    def spiralMatrixIII(self, R, C, r0, c0):
        """
        :type R: int
        :type C: int
        :type r0: int
        :type c0: int
        :rtype: List[List[int]]
        """
        r, c = r0, c0
        result = [[r, c]]
        x, y, n, i = 0, 1, 0, 0
        while len(result) < R*C:
            r, c, i = r+x, c+y, i+1
            if 0 <= r < R and 0 <= c < C:
                result.append([r, c])
            if i == n//2+1:
                x, y, n, i = y, -x, n+1, 0
        return result
```

robot-bounded-in-circle.py

```python
# On an infinite plane, a robot initially stands at (0, 0) and faces north.  The
# robot can receive one of three instructions:
#
#
#       "G": go straight 1 unit;
#       "L": turn 90 degrees to the left;
#       "R": turn 90 degress to the right.
#
#
# The robot performs the instructions given in order, and repeats them forever.
#
# Return true if and only if there exists a circle in the plane such that the
# robot never leaves the circle.
#
#
#
# Example 1:
#
# Input: "GGLLGG"
# Output: true
# Explanation:
# The robot moves from (0,0) to (0,2), turns 180 degrees, and then returns to
# (0,0).
# When repeating these instructions, the robot remains in the circle of radius 2
# centered at the origin.
#
#
# Example 2:
#
# Input: "GG"
# Output: false
# Explanation:
# The robot moves north indefinitely.
#
#
# Example 3:
#
# Input: "GL"
# Output: true
# Explanation:
# The robot moves from (0, 0) -> (0, 1) -> (-1, 1) -> (-1, 0) -> (0, 0) -> ...
#
#
#
#
# Note:
#
#
#       1 <= instructions.length <= 100
#       instructions[i] is in {'G', 'L', 'R'}# Time:  O(n)
# Space: O(1)

class Solution(object):
    def isRobotBounded(self, instructions):
        """
        :type instructions: str
        :rtype: bool
        """
```

```python
directions = [[ 1, 0], [0, -1], [-1, 0], [0, 1]]
x, y, i = 0, 0, 0
for instruction in instructions:
    if instruction == 'R':
        i = (i+1) % 4;
    elif instruction == 'L':
        i = (i-1) % 4;
    else:
        x += directions[i][0]
        y += directions[i][1]
return (x == 0 and y == 0) or i > 0
```

super-pow.py

```python
# Your task is to calculate ab mod 1337 where a is a positive integer and b is
# an extremely large positive integer given in the form of an array.
#
# Example 1:
#
#
# Input: a = 2, b = [3]
# Output: 8
#
#
#
# Example 2:
#
# Input: a = 2, b = [1,0]
# Output: 1024# Time:  O(n), n is the size of b.
# Space: O(1)

class Solution(object):
    def superPow(self, a, b):
        """
        :type a: int
        :type b: List[int]
        :rtype: int
        """
        def myPow(a, n, b):
            result = 1
            x = a % b
            while n:
                if n & 1:
                    result = result * x % b
                n >>= 1
                x = x * x % b
            return result % b

        result = 1
        for digit in b:
            result = myPow(result, 10, 1337) * myPow(a, digit, 1337) % 1337
        return result
```

find-duplicate-subtrees.py

```python
# Given the root of a binary tree, return all duplicate subtrees.
#
# For each kind of duplicate subtrees, you only need to return the root node of
# any one of them.
#
# Two trees are duplicate if they have the same structure with the same node
# values.
#
#
# Example 1:
#
# Input: root = [1,2,3,4,null,2,4,null,null,4]
# Output: [[2,4],[4]]
#
#
# Example 2:
#
# Input: root = [2,1,1]
# Output: [[1]]
#
#
# Example 3:
#
# Input: root = [2,2,2,3,null,3,null]
# Output: [[2,3],[3]]
#
#
#
# Constraints:
#
#
#       The number of the nodes in the tree will be in the range [1, 10^4]
#       -200 <= Node.val <= 200# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def findDuplicateSubtrees(self, root):
        """
        :type root: TreeNode
        :rtype: List[TreeNode]
        """
        def getid(root, lookup, trees):
            if root:
                node_id = lookup[root.val, \
                                 getid(root.left, lookup, trees), \
                                 getid(root.right, lookup, trees)]
                trees[node_id].append(root)
                return node_id
        trees = collections.defaultdict(list)
        lookup = collections.defaultdict()
        lookup.default_factory = lookup.__len__
        getid(root, lookup, trees)
        return [roots[0] for roots in trees.values() if len(roots) > 1]
```

```python
# Time:  O(n * h)
# Space: O(n * h)
class Solution2(object):
    def findDuplicateSubtrees(self, root):
        """
        :type root: TreeNode
        :rtype: List[TreeNode]
        """
        def postOrderTraversal(node, lookup, result):
            if not node:
                return ""
            s = "(" + postOrderTraversal(node.left, lookup, result) + \
                str(node.val) + \
                postOrderTraversal(node.right, lookup, result) + \
                ")"
            if lookup[s] == 1:
                result.append(node)
            lookup[s] += 1
            return s

        lookup = collections.defaultdict(int)
        result = []
        postOrderTraversal(root, lookup, result)
        return result
```

insertion-sort-list.py

```python
# Sort a linked list using insertion sort.
#
#
#
#
#
#
# A graphical example of insertion sort. The partial sorted list (black)
# initially contains only the first element in the list.
#
# With each iteration one element (red) is removed from the input data and
# inserted in-place into the sorted list
#
#
#
#
#
#
# Algorithm of Insertion Sort:
#
#
#       Insertion sort iterates, consuming one input element each repetition,
# and growing a sorted output list.
#       At each iteration, insertion sort removes one element from the input
# data, finds the location it belongs within the sorted list, and inserts it
# there.
#       It repeats until no input elements remain.
#
#
#
#
# Example 1:
#
# Input: 4->2->1->3
# Output: 1->2->3->4
#
#
# Example 2:
#
# Input: -1->5->3->4->0
# Output: -1->0->3->4->5# Time:  O(n ^ 2)
# Space: O(1)

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

    def __repr__(self):
        if self:
            return "{} -> {}".format(self.val, repr(self.next))
        else:
            return "Nil"

class Solution(object):
    # @param head, a ListNode
    # @return a ListNode
    def insertionSortList(self, head):
```

```
        if head is None or self.isSorted(head):
            return head

        dummy = ListNode(-2147483648)
        dummy.next = head
        cur, sorted_tail = head.next, head
        while cur:
            prev = dummy
            while prev.next.val < cur.val:
                prev = prev.next
            if prev == sorted_tail:
                cur, sorted_tail = cur.next, cur
            else:
                cur.next, prev.next, sorted_tail.next = prev.next, cur, cur.next
                cur = sorted_tail.next

        return dummy.next

    def isSorted(self, head):
        while head and head.next:
            if head.val > head.next.val:
                return False
            head = head.next
        return True
```

mirror-reflection.py

```python
# Example 1:
#
# Input: p = 2, q = 1
# Output: 2
# Explanation: The ray meets receptor 2 the first time it gets reflected back to
# the left wall.
#
#
#
# Note:
#
#
#       1 <= p <= 1000
#       0 <= q <= p# Time:  O(1)
# Space: O(1)

class Solution(object):
    def mirrorReflection(self, p, q):
        """
        :type p: int
        :type q: int
        :rtype: int
        """
        # explanation commented in the following solution
        return 2 if (p & -p) > (q & -q) else 0 if (p & -p) < (q & -q) else 1


# Time:  O(log(max(p, q))) = O(1) due to 32-bit integer
# Space: O(1)
class Solution2(object):
    def mirrorReflection(self, p, q):
        """
        :type p: int
        :type q: int
        :rtype: int
        """
        def gcd(a, b):
            while b:
                a, b = b, a % b
            return a

        lcm = p*q // gcd(p, q)
        # let a = lcm / p, b = lcm / q
        if lcm // p % 2 == 1:
            if lcm // q % 2 == 1:
                return 1  # a is odd, b is odd <=> (p & -p) == (q & -q)
            return 2  # a is odd, b is even <=> (p & -p) > (q & -q)
        return 0  # a is even, b is odd <=> (p & -p) < (q & -q)
```

reverse-linked-list-ii.py

```
# Reverse a linked list from position m to n. Do it in one-pass.
#
# Note: 1  m  n  length of list.
#
# Example:
#
# Input: 1->2->3->4->5->NULL, m = 2, n = 4
# Output: 1->4->3->2->5->NULL# Time:  O(n)
# Space: O(1)

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

    def __repr__(self):
        if self:
            return "{} -> {}".format(self.val, repr(self.next))
class Solution(object):
    # @param head, a ListNode
    # @param m, an integer
    # @param n, an integer
    # @return a ListNode
    def reverseBetween(self, head, m, n):
        diff, dummy, cur = n - m + 1, ListNode(-1), head
        dummy.next = head

        last_unswapped = dummy
        while cur and m > 1:
            cur, last_unswapped, m = cur.next, cur, m - 1

        prev, first_swapped = last_unswapped,  cur
        while cur and diff > 0:
            cur.next, prev, cur, diff = prev, cur, cur.next, diff - 1

        last_unswapped.next, first_swapped.next = prev, cur

        return dummy.next
```

validate-binary-tree-nodes.py

```python
# You have n binary tree nodes numbered from 0 to n - 1 where node i has two
# children leftChild[i] and rightChild[i], return true if and only if all the
# given nodes form exactly one valid binary tree.
#
# If node i has no left child then leftChild[i] will equal -1, similarly for the
# right child.
#
# Note that the nodes have no values and that we only use the node numbers in
# this problem.
#
#
# Example 1:
#
#
#
# Input: n = 4, leftChild = [1,-1,3,-1], rightChild = [2,-1,-1,-1]
# Output: true
#
#
# Example 2:
#
#
#
# Input: n = 4, leftChild = [1,-1,3,-1], rightChild = [2,3,-1,-1]
# Output: false
#
#
# Example 3:
#
#
#
# Input: n = 2, leftChild = [1,0], rightChild = [-1,-1]
# Output: false
#
#
# Example 4:
#
#
#
# Input: n = 6, leftChild = [1,-1,-1,4,-1,-1], rightChild = [2,-1,-1,5,-1,-1]
# Output: false
#
#
#
# Constraints:
#
#
#       1 <= n <= 10^4
#       leftChild.length == rightChild.length == n
#       -1 <= leftChild[i], rightChild[i] <= n - 1# Time:  O(n)
# Space: O(n)

class Solution(object):
    def validateBinaryTreeNodes(self, n, leftChild, rightChild):
        """
        :type n: int
        :type leftChild: List[int]
        :type rightChild: List[int]
```

```
    :rtype: bool
    """
    roots = set(range(n)) - set(leftChild) - set(rightChild)
    if len(roots) != 1:
        return False
    root, = roots
    stk = [root]
    lookup = set([root])
    while stk:
        node = stk.pop()
        for c in (leftChild[node], rightChild[node]):
            if c < 0:
                continue
            if c in lookup:
                return False
            lookup.add(c)
            stk.append(c)
    return len(lookup) == n
```

unique-paths-ii.py

```python
# 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).
#
# 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.
#
# Note: m and n will be at most 100.
#
# Example 1:
#
# Input:
# [
#   [0,0,0],
#   [0,1,0],
#   [0,0,0]
# ]
# Output: 2
# Explanation:
# There is one obstacle in the middle of the 3x3 grid above.
# There are two ways to reach the bottom-right corner:
# 1. Right -> Right -> Down -> Down
# 2. Down -> Down -> Right -> Right# Time:  O(m * n)
# Space: O(m + n)

class Solution(object):
    # @param obstacleGrid, a list of lists of integers
    # @return an integer
    def uniquePathsWithObstacles(self, obstacleGrid):
        """
        :type obstacleGrid: List[List[int]]
        :rtype: int
        """
        m, n = len(obstacleGrid), len(obstacleGrid[0])

        ways = [0]*n
        ways[0] = 1
        for i in xrange(m):
            if obstacleGrid[i][0] == 1:
                ways[0] = 0
            for j in xrange(n):
                if obstacleGrid[i][j] == 1:
                    ways[j] = 0
                elif j>0:
                    ways[j] += ways[j-1]
        return ways[-1]
```

longest-absolute-file-path.py

```
# Suppose we have the file system represented in the following picture:
#
#
#
# We will represent the file system as a string where "\n\t" mean a subdirectory
# of the main directory, "\n\t\t" means a subdirectory of the subdirectory of the
# main directory and so on. Each folder will be represented as a string of letters
# and/or digits. Each file will be in the form "s1.s2" where s1 and s2 are strings
# of letters and/or digits.
#
# For example, the file system above is represented as "dir\n\tsubdir1\n\t\tfile
# 1.ext\n\t\tsubsubdir1\n\tsubdir2\n\t\tsubsubdir2\n\t\t\tfile2.ext".
#
# Given a string input representing the file system in the explained format,
# return the length of the longest absolute path to a file in the abstracted file
# system. If there is no file in the system, return 0.
#
#
# Example 1:
#
# Input: input = "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext"
# Output: 20
# Explanation: We have only one file and its path is "dir/subdir2/file.ext" of
# length 20.
# The path "dir/subdir1" doesn't contain any files.
#
#
# Example 2:
#
# Input: input = "dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\tsubdir2\n\t\t
# subsubdir2\n\t\t\tfile2.ext"
# Output: 32
# Explanation: We have two files:
# "dir/subdir1/file1.ext" of length 21
# "dir/subdir2/subsubdir2/file2.ext" of length 32.
# We return 32 since it is the longest path.
#
#
# Example 3:
#
# Input: input = "a"
# Output: 0
# Explanation: We don't have any files.
#
#
#
# Constraints:
#
#
#       1 <= input.length <= 104
#       input may contain lower-case or upper-case English letters, a new line
# character '\n', a tab character '\t', a dot '.', a space ' ' or digits.# Time:  O(n)
# Space: O(d), d is the max depth of the paths

class Solution(object):
    def lengthLongestPath(self, input):
        """
        :type input: str
```

```
:rtype: int
"""
def split_iter(s, tok):
    start = 0
    for i in xrange(len(s)):
        if s[i] == tok:
            yield s[start:i]
            start = i + 1
    yield s[start:]


max_len = 0
path_len = {0: 0}
for line in split_iter(input, '\n'):
    name = line.lstrip('\t')
    depth = len(line) - len(name)
    if '.' in name:
        max_len = max(max_len, path_len[depth] + len(name))
    else:
        path_len[depth + 1] = path_len[depth] + len(name) + 1
return max_len
```

vowel-spellchecker.py

```
# Given a wordlist, we want to implement a spellchecker that converts a query
# word into a correct word.
#
# For a given query word, the spell checker handles two categories of spelling
# mistakes:
#
#
#       Capitalization: If the query matches a word in the wordlist (case-
# insensitive), then the query word is returned with the same case as the case in
# the wordlist.
#
#
#               Example: wordlist = ["yellow"], query = "YellOw": correct =
# "yellow"
#               Example: wordlist = ["Yellow"], query = "yellow": correct =
# "Yellow"
#               Example: wordlist = ["yellow"], query = "yellow": correct =
# "yellow"
#
#
#       Vowel Errors: If after replacing the vowels ('a', 'e', 'i', 'o', 'u') of
# the query word with any vowel individually, it matches a word in the wordlist
# (case-insensitive), then the query word is returned with the same case as the
# match in the wordlist.
#
#               Example: wordlist = ["YellOw"], query = "yollow": correct =
# "YellOw"
#               Example: wordlist = ["YellOw"], query = "yeellow": correct = ""
# (no match)
#               Example: wordlist = ["YellOw"], query = "yllw": correct = "" (no
# match)
#
#
#
#
# In addition, the spell checker operates under the following precedence rules:
#
#
#       When the query exactly matches a word in the wordlist (case-sensitive),
# you should return the same word back.
#       When the query matches a word up to capitlization, you should return the
# first such match in the wordlist.
#       When the query matches a word up to vowel errors, you should return the
# first such match in the wordlist.
#       If the query has no matches in the wordlist, you should return the empty
# string.
#
#
# Given some queries, return a list of words answer, where answer[i] is the
# correct word for query = queries[i].
#
#
#
# Example 1:
#
# Input: wordlist = ["KiTe","kite","hare","Hare"], queries =
# ["kite","Kite","KiTe","Hare","HARE","Hear","hear","keti","keet","keto"]
# Output: ["kite","KiTe","KiTe","Hare","hare","","","KiTe","","KiTe"]
```

```python
#
#
#
# Note:
#
#
#       1 <= wordlist.length <= 5000
#       1 <= queries.length <= 5000
#       1 <= wordlist[i].length <= 7
#       1 <= queries[i].length <= 7
#       All strings in wordlist and queries consist only of english letters.# Time:  O(n)
# Space: O(w)

class Solution(object):
    def spellchecker(self, wordlist, queries):
        """
        :type wordlist: List[str]
        :type queries: List[str]
        :rtype: List[str]
        """
        vowels = set(['a', 'e', 'i', 'o', 'u'])
        def todev(word):
            return "".join('*' if c.lower() in vowels else c.lower()
                           for c in word)

        words = set(wordlist)
        caps = {}
        vows = {}

        for word in wordlist:
            caps.setdefault(word.lower(), word)
            vows.setdefault(todev(word), word)

        def check(query):
            if query in words:
                return query
            lower = query.lower()
            if lower in caps:
                return caps[lower]
            devow = todev(lower)
            if devow in vows:
                return vows[devow]
            return ""
        return map(check, queries)
```

shuffle-an-array.py

```python
# Shuffle a set of numbers without duplicates.
#
#
# Example:
# // Init an array with set 1, 2, and 3.
# int[] nums = {1,2,3};
# Solution solution = new Solution(nums);
#
# // Shuffle the array [1,2,3] and return its result. Any permutation of [1,2,3]
# must equally likely to be returned.
# solution.shuffle();
#
# // Resets the array back to its original configuration [1,2,3].
# solution.reset();
#
# // Returns the random shuffling of array [1,2,3].
# solution.shuffle();# Time:  O(n)
# Space: O(n)

import random


class Solution(object):

    def __init__(self, nums):
        """

        :type nums: List[int]
        :type size: int
        """
        self.__nums = nums


    def reset(self):
        """
        Resets the array to its original configuration and return it.
        :rtype: List[int]
        """
        return self.__nums


    def shuffle(self):
        """
        Returns a random shuffling of the array.
        :rtype: List[int]
        """
        nums = list(self.__nums)
        for i in xrange(len(nums)):
            j = random.randint(i, len(nums)-1)
            nums[i], nums[j] = nums[j], nums[i]
        return nums
```

minimum-falling-path-sum.py

```python
# Given a square array of integers A, we want the minimum sum of a falling path
# through A.
#
# A falling path starts at any element in the first row, and chooses one element
# from each row.  The next row's choice must be in a column that is different from
# the previous row's column by at most one.
#
#
#
# Example 1:
#
# Input: [[1,2,3],[4,5,6],[7,8,9]]
# Output: 12
# Explanation:
# The possible falling paths are:
#
#
#
#       [1,4,7], [1,4,8], [1,5,7], [1,5,8], [1,5,9]
#       [2,4,7], [2,4,8], [2,5,7], [2,5,8], [2,5,9], [2,6,8], [2,6,9]
#       [3,5,7], [3,5,8], [3,5,9], [3,6,8], [3,6,9]
#
#
# The falling path with the smallest sum is [1,4,7], so the answer is 12.
#
#
# Constraints:
#
#
#       1 <= A.length == A[0].length <= 100
#       -100 <= A[i][j] <= 100# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    def minFallingPathSum(self, A):
        """
        :type A: List[List[int]]
        :rtype: int
        """
        for i in xrange(1, len(A)):
            for j in xrange(len(A[i])):
                A[i][j] += min(A[i-1][max(j-1, 0):j+2])
        return min(A[-1])
```

boats-to-save-people.py

```python
# The i-th person has weight people[i], and each boat can carry a maximum weight
# of limit.
#
# Each boat carries at most 2 people at the same time, provided the sum of
# the weight of those people is at most limit.
#
# Return the minimum number of boats to carry every given person.  (It is
# guaranteed each person can be carried by a boat.)
#
#
#
#
# Example 1:
#
# Input: people = [1,2], limit = 3
# Output: 1
# Explanation: 1 boat (1, 2)
#
#
#
# Example 2:
#
# Input: people = [3,2,2,1], limit = 3
# Output: 3
# Explanation: 3 boats (1, 2), (2) and (3)
#
#
#
# Example 3:
#
# Input: people = [3,5,3,4], limit = 5
# Output: 4
# Explanation: 4 boats (3), (3), (4), (5)
#
# Note:
#
#
#       1 <= people.length <= 50000
#       1 <= people[i] <= limit <= 30000# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def numRescueBoats(self, people, limit):
        """
        :type people: List[int]
        :type limit: int
        :rtype: int
        """
        people.sort()
        result = 0
        left, right = 0, len(people)-1
        while left <= right:
            result += 1
            if people[left] + people[right] <= limit:
                left += 1
            right -= 1
        return result
```

sum-root-to-leaf-numbers.py

```
# 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.
#
# Note: A leaf is a node with no children.
#
# Example:
#
# Input: [1,2,3]
#     1
#    / \
#   2   3
# Output: 25
# Explanation:
# The root-to-leaf path 1->2 represents the number 12.
# The root-to-leaf path 1->3 represents the number 13.
# Therefore, sum = 12 + 13 = 25.
#
# Example 2:
#
# Input: [4,9,0,5,1]
#     4
#    / \
#   9   0
#  / \
# 5   1
# Output: 1026
# Explanation:
# The root-to-leaf path 4->9->5 represents the number 495.
# The root-to-leaf path 4->9->1 represents the number 491.
# The root-to-leaf path 4->0 represents the number 40.
# Therefore, sum = 495 + 491 + 40 = 1026.# Time:  O(n)
# Space: O(h), h is height of binary tree

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    # @param root, a tree node
    # @return an integer
    def sumNumbers(self, root):
        return self.sumNumbersRecu(root, 0)

    def sumNumbersRecu(self, root, num):
        if root is None:
            return 0

        if root.left is None and root.right is None:
            return num * 10 + root.val

        return self.sumNumbersRecu(root.left, num * 10 + root.val) + self.sumNumbersRecu(root.right, num * 10
```

different-ways-to-add-parentheses.py

```python
# 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"
# Output: [0, 2]
# Explanation:
# ((2-1)-1) = 0
# (2-(1-1)) = 2
#
# Example 2:
#
# Input: "2*3-4*5"
# Output: [-34, -14, -10, -10, 10]
# Explanation:
# (2*(3-(4*5))) = -34
# ((2*3)-(4*5)) = -14
# ((2*(3-4))*5) = -10
# (2*((3-4)*5)) = -10
# (((2*3)-4)*5) = 10# Time:  O(n * 4^n / n^(3/2)) ~= n * Catalan numbers = n * (C(2n, n) - C(2n, n - 1)),
#                                 due to the size of the results is Catalan numbers,
#                                 and every way of evaluation is the length of the string,
#                                 so the time complexity is at most n * Catalan numbers.
# Space: O(n * 4^n / n^(3/2)), the cache size of lookup is at most n * Catalan numbers.

import operator
import re


class Solution(object):
    # @param {string} input
    # @return {integer[]}
    def diffWaysToCompute(self, input):
        tokens = re.split('(\D)', input)
        nums = map(int, tokens[::2])
        ops = map({'+': operator.add, '-': operator.sub, '*': operator.mul}.get, tokens[1::2])
        lookup = [[None for _ in xrange(len(nums))] for _ in xrange(len(nums))]

        def diffWaysToComputeRecu(left, right):
            if left == right:
                return [nums[left]]
            if lookup[left][right]:
                return lookup[left][right]
            lookup[left][right] = [ops[i](x, y)
                                   for i in xrange(left, right)
                                   for x in diffWaysToComputeRecu(left, i)
                                   for y in diffWaysToComputeRecu(i + 1, right)]
            return lookup[left][right]

        return diffWaysToComputeRecu(0, len(nums) - 1)

class Solution2(object):
    # @param {string} input
    # @return {integer[]}
    def diffWaysToCompute(self, input):
        lookup = [[None for _ in xrange(len(input) + 1)] for _ in xrange(len(input) + 1)]
```

```
ops = {'+': operator.add, '-': operator.sub, '*': operator.mul}

def diffWaysToComputeRecu(left, right):
    if lookup[left][right]:
        return lookup[left][right]
    result = []
    for i in xrange(left, right):
        if input[i] in ops:
            for x in diffWaysToComputeRecu(left, i):
                for y in diffWaysToComputeRecu(i + 1, right):
                    result.append(ops[input[i]](x, y))

    if not result:
        result = [int(input[left:right])]
    lookup[left][right] = result
    return lookup[left][right]

return diffWaysToComputeRecu(0, len(input))
```

as-far-from-land-as-possible.py

```python
# Given an N x N grid containing only values 0 and 1, where 0 represents
# water and 1 represents land, find a water cell such that its distance to the
# nearest land cell is maximized and return the distance.
#
# The distance used in this problem is the Manhattan distance: the distance
# between two cells (x0, y0) and (x1, y1) is |x0 - x1| + |y0 - y1|.
#
# If no land or water exists in the grid, return -1.
#
#
#
# Example 1:
#
#
#
# Input: [[1,0,1],[0,0,0],[1,0,1]]
# Output: 2
# Explanation:
# The cell (1, 1) is as far as possible from all the land with distance 2.
#
#
# Example 2:
#
#
#
# Input: [[1,0,0],[0,0,0],[0,0,0]]
# Output: 4
# Explanation:
# The cell (2, 2) is as far as possible from all the land with distance 4.
#
#
#
#
# Note:
#
#
#       1 <= grid.length == grid[0].length <= 100
#       grid[i][j] is 0 or 1# Time:  O(m * n)
# Space: O(m * n)

import collections


class Solution(object):
    def maxDistance(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        q = collections.deque([(i, j) for i in xrange(len(grid))
                                      for j in xrange(len(grid[0])) if grid[i][j] == 1])
        if len(q) == len(grid)*len(grid[0]):
            return -1
        level = -1
        while q:
            next_q = collections.deque()
            while q:
```

```
        x, y = q.popleft()
        for dx, dy in directions:
            nx, ny = x+dx, y+dy
            if not (0 <= nx < len(grid) and
                    0 <= ny < len(grid[0]) and
                    grid[nx][ny] == 0):
                continue
            next_q.append((nx, ny))
            grid[nx][ny] = 1
    q = next_q
    level += 1
return level
```

broken-calculator.py

```python
# On a broken calculator that has a number showing on its display, we can
# perform two operations:
#
#
#       Double: Multiply the number on the display by 2, or;
#       Decrement: Subtract 1 from the number on the display.
#
#
# Initially, the calculator is displaying the number X.
#
# Return the minimum number of operations needed to display the number Y.
#
#
#
# Example 1:
#
# Input: X = 2, Y = 3
# Output: 2
# Explanation: Use double operation and then decrement operation {2 -> 4 -> 3}.
#
#
# Example 2:
#
# Input: X = 5, Y = 8
# Output: 2
# Explanation: Use decrement and then double {5 -> 4 -> 8}.
#
#
# Example 3:
#
# Input: X = 3, Y = 10
# Output: 3
# Explanation:  Use double, decrement and double {3 -> 6 -> 5 -> 10}.
#
#
# Example 4:
#
# Input: X = 1024, Y = 1
# Output: 1023
# Explanation: Use decrement operations 1023 times.
#
#
#
#
# Note:
#
#
#       1 <= X <= 10^9
#       1 <= Y <= 10^9# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def brokenCalc(self, X, Y):
        """
        :type X: int
        :type Y: int
        :rtype: int
        """
```

```python
result = 0
while X < Y:
    if Y%2:
        Y += 1
    else:
        Y /= 2
    result += 1
return result + X-Y
```

construct-binary-tree-from-preorder-and-inorder-traversal.py

```
# Given preorder and inorder traversal of a tree, construct the binary tree.
#
# Note:
#
# You may assume that duplicates do not exist in the tree.
#
# For example, given
#
# preorder = [3,9,20,15,7]
# inorder = [9,3,15,20,7]
#
# Return the following binary tree:
#
#     3
#    / \
#   9  20
#     /  \
#    15   7# Time:  O(n)
# Space: O(n)

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    # @param preorder, a list of integers
    # @param inorder, a list of integers
    # @return a tree node
    def buildTree(self, preorder, inorder):
        lookup = {}
        for i, num in enumerate(inorder):
            lookup[num] = i
        return self.buildTreeRecu(lookup, preorder, inorder, 0, 0, len(inorder))

    def buildTreeRecu(self, lookup, preorder, inorder, pre_start, in_start, in_end):
        if in_start == in_end:
            return None
        node = TreeNode(preorder[pre_start])
        i = lookup[preorder[pre_start]]
        node.left = self.buildTreeRecu(lookup, preorder, inorder, pre_start + 1, in_start, i)
        node.right = self.buildTreeRecu(lookup, preorder, inorder, pre_start + 1 + i - in_start, i + 1, in_end
        return node


# time: O(n)
# space: O(n)
class Solution2(object):
    def buildTree(self, preorder, inorder):
        """
        :type preorder: List[int]
        :type inorder: List[int]
        :rtype: TreeNode
        """
        preorder_iterator = iter(preorder)
        inorder_lookup = {n: i for i, n in enumerate(inorder)}
```

```python
def helper(start, end):
    if start > end:
        return None

    root_val = next(preorder_iterator)
    root = TreeNode(root_val)
    idx = inorder_lookup[root_val]
    root.left = helper(start, idx-1)
    root.right = helper(idx+1, end)
    return root

return helper(0, len(inorder)-1)
```

longest-palindromic-substring.py

```python
# Given a string s, find the longest palindromic substring in s. You may assume
# that the maximum length of s is 1000.
#
# Example 1:
#
# Input: "babad"
# Output: "bab"
# Note: "aba" is also a valid answer.
#
#
# Example 2:
#
# Input: "cbbd"
# Output: "bb"# Time:  O(n)
# Space: O(n)

class Solution(object):
    def longestPalindrome(self, s):
        """
        :type s: str
        :rtype: str
        """
        def preProcess(s):
            if not s:
                return ['^', '$']
            T = ['^']
            for c in s:
                T +=  ['#', c]
            T += ['#', '$']
            return T

        T = preProcess(s)
        P = [0] * len(T)
        center, right = 0, 0
        for i in xrange(1, len(T) - 1):
            i_mirror = 2 * center - i
            if right > i:
                P[i] = min(right - i, P[i_mirror])
            else:
                P[i] = 0

            while T[i + 1 + P[i]] == T[i - 1 - P[i]]:
                P[i] += 1

            if i + P[i] > right:
                center, right = i, i + P[i]

        max_i = 0
        for i in xrange(1, len(T) - 1):
            if P[i] > P[max_i]:
                max_i = i
        start = (max_i - 1 - P[max_i]) / 2
        return s[start : start + P[max_i]]
```

replace-words.py

```python
# In English, we have a concept called root, which can be followed by some other
# words to form another longer word - let's call this word successor. For example,
# when the root "an" is followed by the successor word "other", we can form a new
# word "another".
#
# Given a dictionary consisting of many roots and a sentence consisting of words
# separted by spaces. You need to replace all the successors in the sentence with
# the root forming it. If a successor can be replaced by more than one
# root, replace it with the root with the shortest length.
#
# Return the sentence after the replacement.
#
#
# Example 1:
# Input: dictionary = ["cat","bat","rat"], sentence = "the cattle was rattled by
# the battery"
# Output: "the cat was rat by the bat"
# Example 2:
# Input: dictionary = ["a","b","c"], sentence = "aadsfasf absbs bbab cadsfafs"
# Output: "a a b c"
# Example 3:
# Input: dictionary = ["a", "aa", "aaa", "aaaa"], sentence = "a aa a aaaa aaa
# aaa aaa aaaaaa bbb baba ababa"
# Output: "a a a a a a a a bbb baba a"
# Example 4:
# Input: dictionary = ["catt","cat","bat","rat"], sentence = "the cattle was
# rattled by the battery"
# Output: "the cat was rat by the bat"
# Example 5:
# Input: dictionary = ["ac","ab"], sentence = "it is abnormal that this solution
# is accepted"
# Output: "it is ab that this solution is ac"
#
#
# Constraints:
#
#
#       1 <= dictionary.length <= 1000
#       1 <= dictionary[i].length <= 100
#       dictionary[i] consists of only lower-case letters.
#       1 <= sentence.length <= 10^6
#       sentence consists of only lower-case letters ans spaces.
#       The number of words in sentence is in the range [1, 1000]
#       The length of each word in sentence is in the range [1, 1000]
#       Each two words in sentence will be separted by exactly one space.
#       sentence doesn't have leading or trailing spaces.# Time:  O(n)
# Space: O(t), t is the number of nodes in trie

import collections


class Solution(object):
    def replaceWords(self, dictionary, sentence):
        """
        :type dictionary: List[str]
        :type sentence: str
        :rtype: str
        """
```

```python
_trie = lambda: collections.defaultdict(_trie)
trie = _trie()
for word in dictionary:
    reduce(dict.__getitem__, word, trie).setdefault("_end")

def replace(word):
    curr = trie
    for i, c in enumerate(word):
        if c not in curr:
            break
        curr = curr[c]
        if "_end" in curr:
            return word[:i+1]
    return word

return " ".join(map(replace, sentence.split()))
```

k-concatenation-maximum-sum.py

```python
# Given an integer array arr and an integer k, modify the array by repeating it
# k times.
#
# For example, if arr = [1, 2] and k = 3 then the modified array will be [1, 2,
# 1, 2, 1, 2].
#
# Return the maximum sub-array sum in the modified array. Note that the length
# of the sub-array can be 0 and its sum in that case is 0.
#
# As the answer can be very large, return the answer modulo 10^9 + 7.
#
#
# Example 1:
#
# Input: arr = [1,2], k = 3
# Output: 9
#
#
# Example 2:
#
# Input: arr = [1,-2,1], k = 5
# Output: 2
#
#
# Example 3:
#
# Input: arr = [-1,-2], k = 7
# Output: 0
#
#
#
# Constraints:
#
#
#       1 <= arr.length <= 10^5
#       1 <= k <= 10^5
#       -10^4 <= arr[i] <= 10^4# Time:  O(n)
# Space: O(1)

class Solution(object):
    def kConcatenationMaxSum(self, arr, k):
        """
        :type arr: List[int]
        :type k: int
        :rtype: int
        """
        def max_sub_k_array(arr, k):
            result, curr = float("-inf"), float("-inf")
            for _ in xrange(k):
                for x in arr:
                    curr = max(curr+x, x)
                    result = max(result, curr)
            return result

        MOD = 10**9+7
        if k == 1:
            return max(max_sub_k_array(arr, 1), 0) % MOD
        return (max(max_sub_k_array(arr, 2), 0) + (k-2)*max(sum(arr), 0)) % MOD
```

ones-and-zeroes.py

```python
# Given an array, strs, with strings consisting of only 0s and 1s. Also two
# integers m and n.
#
# Now your task is to find the maximum number of strings that you can form with
# given m 0s and n 1s. Each 0 and 1 can be used at most once.
#
#
# Example 1:
#
# Input: strs = ["10","0001","111001","1","0"], m = 5, n = 3
# Output: 4
# Explanation: This are totally 4 strings can be formed by the using of 5 0s and
# 3 1s, which are "10","0001","1","0".
#
#
# Example 2:
#
# Input: strs = ["10","0","1"], m = 1, n = 1
# Output: 2
# Explanation: You could form "10", but then you'd have nothing left. Better
# form "0" and "1".
#
#
#
# Constraints:
#
#
#       1 <= strs.length <= 600
#       1 <= strs[i].length <= 100
#       strs[i] consists only of digits '0' and '1'.
#       1 <= m, n <= 100# Time:  O(s * m * n), s is the size of the array.
# Space: O(m * n)

class Solution(object):
    def findMaxForm(self, strs, m, n):
        """
        :type strs: List[str]
        :type m: int
        :type n: int
        :rtype: int
        """
        dp = [[0 for _ in xrange(n+1)] for _ in xrange(m+1)]
        for s in strs:
            zero_count, one_count = 0, 0
            for c in s:
                if c == '0':
                    zero_count += 1
                elif c == '1':
                    one_count += 1

            for i in reversed(xrange(zero_count, m+1)):
                for j in reversed(xrange(one_count, n+1)):
                    dp[i][j] = max(dp[i][j], dp[i-zero_count][j-one_count]+1)
        return dp[m][n]
```

validate-binary-search-tree.py

```python
# 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:
#
#     2
#    / \
#   1   3
#
# Input: [2,1,3]
# Output: true
#
#
# Example 2:
#
#     5
#    / \
#   1   4
#      / \
#     3   6
#
# Input: [5,1,4,null,null,3,6]
# Output: false
# Explanation: The root node's value is 5 but its right child's value is 4.# Time:  O(n)
# Space: O(1)

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

# Morris Traversal Solution
class Solution(object):
    # @param root, a tree node
    # @return a list of integers
    def isValidBST(self, root):
        prev, cur = None, root
        while cur:
            if cur.left is None:
                if prev and prev.val >= cur.val:
                    return False
                prev = cur
                cur = cur.right
            else:
                node = cur.left
                while node.right and node.right != cur:
```

```python
                node = node.right

            if node.right is None:
                node.right = cur
                cur = cur.left
            else:
                if prev and prev.val >= cur.val:
                    return False
                node.right = None
                prev = cur
                cur = cur.right

        return True


# Time:  O(n)
# Space: O(h)
class Solution2(object):
    # @param root, a tree node
    # @return a boolean
    def isValidBST(self, root):
        return self.isValidBSTRecu(root, float("-inf"), float("inf"))

    def isValidBSTRecu(self, root, low, high):
        if root is None:
            return True

        return low < root.val and root.val < high \
            and self.isValidBSTRecu(root.left, low, root.val) \
            and self.isValidBSTRecu(root.right, root.val, high)
```

decoded-string-at-index.py

```
# An encoded string S is given.  To find and write the decoded string to a tape,
# the encoded string is read one character at a time and the following steps are
# taken:
#
#
#       If the character read is a letter, that letter is written onto the tape.
#       If the character read is a digit (say d), the entire current tape is
# repeatedly written d-1 more times in total.
#
#
# Now for some encoded string S, and an index K, find and return the K-th letter
# (1 indexed) in the decoded string.
#
#
#
#
# Example 1:
#
# Input: S = "leet2code3", K = 10
# Output: "o"
# Explanation:
# The decoded string is "leetleetcodeleetleetcodeleetleetcode".
# The 10th letter in the string is "o".
#
#
#
# Example 2:
#
# Input: S = "ha22", K = 5
# Output: "h"
# Explanation:
# The decoded string is "hahahaha".  The 5th letter is "h".
#
#
#
# Example 3:
#
# Input: S = "a2345678999999999999999", K = 1
# Output: "a"
# Explanation:
# The decoded string is "a" repeated 8301530446056247680 times.  The 1st letter
# is "a".
#
#
#
#
#
#
# Constraints:
#
#
#       2 <= S.length <= 100
#       S will only contain lowercase letters and digits 2 through 9.
#       S starts with a letter.
#       1 <= K <= 10^9
#       It's guaranteed that K is less than or equal to the length of the
# decoded string.
#       The decoded string is guaranteed to have less than 2^63 letters.# Time:  O(n)
```

```python
# Space: O(1)

class Solution(object):
    def decodeAtIndex(self, S, K):
        """
        :type S: str
        :type K: int
        :rtype: str
        """
        i = 0
        for c in S:
            if c.isdigit():
                i *= int(c)
            else:
                i += 1

        for c in reversed(S):
            K %= i
            if K == 0 and c.isalpha():
                return c

            if c.isdigit():
                i /= int(c)
            else:
                i -= 1
```

rotate-list.py

```
# Given a linked list, rotate the list to the right by k places, where k is non-
# negative.
#
# Example 1:
#
# Input: 1->2->3->4->5->NULL, k = 2
# Output: 4->5->1->2->3->NULL
# Explanation:
# rotate 1 steps to the right: 5->1->2->3->4->NULL
# rotate 2 steps to the right: 4->5->1->2->3->NULL
#
#
# Example 2:
#
# Input: 0->1->2->NULL, k = 4
# Output: 2->0->1->NULL
# Explanation:
# rotate 1 steps to the right: 2->0->1->NULL
# rotate 2 steps to the right: 1->2->0->NULL
# rotate 3 steps to the right: 0->1->2->NULL
# rotate 4 steps to the right: 2->0->1->NULL# Time:  O(n)
# Space: O(1)

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

    def __repr__(self):
        if self:
            return "{} -> {}".format(self.val, repr(self.next))

class Solution(object):
    def rotateRight(self, head, k):
        """
        :type head: ListNode
        :type k: int
        :rtype: ListNode
        """
        if not head or not head.next:
            return head

        n, cur = 1, head
        while cur.next:
            cur = cur.next
            n += 1
        cur.next = head

        cur, tail = head, cur
        for _ in xrange(n - k % n):
            tail = cur
            cur = cur.next
        tail.next = None

        return cur
```

minimum-path-sum.py

```python
# 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.
#
# Example:
#
# Input:
# [
#   [1,3,1],
#   [1,5,1],
#   [4,2,1]
# ]
# Output: 7
# Explanation: Because the path 1→3→1→1→1 minimizes the sum.# Time:  O(m * n)
# Space: O(m + n)

class Solution(object):
    # @param grid, a list of lists of integers
    # @return an integer
    def minPathSum(self, grid):
        sum = list(grid[0])
        for j in xrange(1, len(grid[0])):
            sum[j] = sum[j - 1] + grid[0][j]

        for i in xrange(1, len(grid)):
            sum[0] += grid[i][0]
            for j in xrange(1, len(grid[0])):
                sum[j] = min(sum[j - 1], sum[j]) + grid[i][j]

        return sum[-1]
```

spiral-matrix-ii.py

```
# Given a positive integer n, generate a square matrix filled with elements from
# 1 to n2 in spiral order.
#
# Example:
#
# Input: 3
# Output:
# [
#  [ 1, 2, 3 ],
#  [ 8, 9, 4 ],
#  [ 7, 6, 5 ]
# ]# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    # @return a list of lists of integer
    def generateMatrix(self, n):
        matrix = [[0 for _ in xrange(n)] for _ in xrange(n)]

        left, right, top, bottom, num = 0, n - 1, 0, n - 1, 1

        while left <= right and top <= bottom:
            for j in xrange(left, right + 1):
                matrix[top][j] = num
                num += 1
            for i in xrange(top + 1, bottom):
                matrix[i][right] = num
                num += 1
            for j in reversed(xrange(left, right + 1)):
                if top < bottom:
                    matrix[bottom][j] = num
                    num += 1
            for i in reversed(xrange(top + 1, bottom)):
                if left < right:
                    matrix[i][left] = num
                    num += 1
            left, right, top, bottom = left + 1, right - 1, top + 1, bottom - 1

        return matrix
```

palindromic-substrings.py

```python
# Given a string, your task is to count how many palindromic substrings in this
# string.
#
# The substrings with different start indexes or end indexes are counted as
# different substrings even they consist of same characters.
#
# Example 1:
#
# Input: "abc"
# Output: 3
# Explanation: Three palindromic strings: "a", "b", "c".
#
#
#
#
# Example 2:
#
# Input: "aaa"
# Output: 6
# Explanation: Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".
#
#
#
#
# Note:
#
#
#       The input string length won't exceed 1000.# Time:  O(n)
# Space: O(n)

class Solution(object):
    def countSubstrings(self, s):
        """
        :type s: str
        :rtype: int
        """
        def manacher(s):
            s = '^#' + '#'.join(s) + '#$'
            P = [0] * len(s)
            C, R = 0, 0
            for i in xrange(1, len(s) - 1):
                i_mirror = 2*C-i
                if R > i:
                    P[i] = min(R-i, P[i_mirror])
                while s[i+1+P[i]] == s[i-1-P[i]]:
                    P[i] += 1
                if i+P[i] > R:
                    C, R = i, i+P[i]
            return P
        return sum((max_len+1)//2 for max_len in manacher(s))
```

numbers-with-same-consecutive-differences.py

```python
# Example 2:
#
# Input: N = 2, K = 1
# Output: [10,12,21,23,32,34,43,45,54,56,65,67,76,78,87,89,98]# Time:  O(2^n)
# Space: O(2^n)

class Solution(object):
    def numsSameConsecDiff(self, N, K):
        """
        :type N: int
        :type K: int
        :rtype: List[int]
        """
        curr = range(10)
        for i in xrange(N-1):
            curr = [x*10 + y for x in curr for y in set([x%10 + K, x%10 - K])
                    if x and 0 <= y < 10]
        return curr
```

longest-well-performing-interval.py

```python
# We are given hours, a list of the number of hours worked per day for a given
# employee.
#
# A day is considered to be a tiring day if and only if the number of hours
# worked is (strictly) greater than 8.
#
# A well-performing interval is an interval of days for which the number of
# tiring days is strictly larger than the number of non-tiring days.
#
# Return the length of the longest well-performing interval.
#
#
# Example 1:
#
# Input: hours = [9,9,6,0,6,6,9]
# Output: 3
# Explanation: The longest well-performing interval is [9,9,6].
#
#
#
# Constraints:
#
#
#       1 <= hours.length <= 10000
#       0 <= hours[i] <= 16# Time:  O(n)
# Space: O(n)

class Solution(object):
    def longestWPI(self, hours):
        """
        :type hours: List[int]
        :rtype: int
        """
        result, accu = 0, 0
        lookup = {}
        for i, h in enumerate(hours):
            accu = accu+1 if h > 8 else accu-1
            if accu > 0:
                result = i+1
            elif accu-1 in lookup:
                # lookup[accu-1] is the leftmost idx with smaller accu,
                # because for i from 1 to some positive k,
                # lookup[accu-i] is a strickly increasing sequence
                result = max(result, i-lookup[accu-1])
            lookup.setdefault(accu, i)
        return result
```

add-two-numbers-ii.py

```python
# You are given two non-empty linked lists representing two non-negative
# integers. The most significant digit comes first and each of their nodes contain
# a single digit. Add the two numbers and return it as a linked list.
#
# You may assume the two numbers do not contain any leading zero, except the
# number 0 itself.
#
# Follow up:
#
# What if you cannot modify the input lists? In other words, reversing the lists
# is not allowed.
#
#
#
# Example:
# Input: (7 -> 2 -> 4 -> 3) + (5 -> 6 -> 4)
# Output: 7 -> 8 -> 0 -> 7# Time:  O(m + n)
# Space: O(m + n)

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None


class Solution(object):
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        stk1, stk2 = [], []
        while l1:
            stk1.append(l1.val)
            l1 = l1.next
        while l2:
            stk2.append(l2.val)
            l2 = l2.next

        prev, head = None, None
        sum = 0
        while stk1 or stk2:
            sum /= 10
            if stk1:
                sum += stk1.pop()
            if stk2:
                sum += stk2.pop()

            head = ListNode(sum % 10)
            head.next = prev
            prev = head

        if sum >= 10:
            head = ListNode(sum / 10)
            head.next = prev

        return head
```

minimum-area-rectangle-ii.py

```python
# Example 4:
#
#
#
# Input: [[3,1],[1,1],[0,1],[2,1],[3,3],[3,2],[0,2],[2,3]]
# Output: 2.00000
# Explanation: The minimum area rectangle occurs at [2,1],[2,3],[3,3],[3,1],
# with an area of 2.# Time:  O(n^2) ~ O(n^3)
# Space: O(n^2)

import collections
import itertools


class Solution(object):
    def minAreaFreeRect(self, points):
        """
        :type points: List[List[int]]
        :rtype: float
        """
        points.sort()
        points = [complex(*z) for z in points]
        lookup = collections.defaultdict(list)
        for P, Q in itertools.combinations(points, 2):
            lookup[P-Q].append((P+Q) / 2)

        result = float("inf")
        for A, candidates in lookup.iteritems():
            for P, Q in itertools.combinations(candidates, 2):
                if A.real * (P-Q).real + A.imag * (P-Q).imag == 0.0:
                    result = min(result, abs(A) * abs(P-Q))
        return result if result < float("inf") else 0.0
```

clumsy-factorial.py

```
# Normally, the factorial of a positive integer n is the product of all positive
# integers less than or equal to n.  For example, factorial(10) = 10 * 9 * 8 * 7 *
# 6 * 5 * 4 * 3 * 2 * 1.
#
# We instead make a clumsy factorial: using the integers in decreasing order,
# we swap out the multiply operations for a fixed rotation of operations: multiply
# (*), divide (/), add (+) and subtract (-) in this order.
#
# For example, clumsy(10) = 10 * 9 / 8 + 7 - 6 * 5 / 4 + 3 - 2 * 1.  However,
# these operations are still applied using the usual order of operations of
# arithmetic: we do all multiplication and division steps before any addition or
# subtraction steps, and multiplication and division steps are processed left to
# right.
#
# Additionally, the division that we use is floor division such that 10 * 9 /
# 8 equals 11.  This guarantees the result is an integer.
#
# Implement the clumsy function as defined above: given an integer N, it returns
# the clumsy factorial of N.
#
#
#
# Example 1:
#
# Input: 4
# Output: 7
# Explanation: 7 = 4 * 3 / 2 + 1
#
#
# Example 2:
#
# Input: 10
# Output: 12
# Explanation: 12 = 10 * 9 / 8 + 7 - 6 * 5 / 4 + 3 - 2 * 1
#
#
#
#
# Note:
#
#
#       1 <= N <= 10000
#       -2^31 <= answer <= 2^31 - 1  (The answer is guaranteed to fit within a
# 32-bit integer.)# Time:  O(1)
# Space: O(1)

# observation:
# i*(i-1)/(i-2) = i+1+2/(i-2)
#     if i = 3  => i*(i-1)/(i-2) = i + 3
#     if i = 4  => i*(i-1)/(i-2) = i + 2
#     if i >= 5 => i*(i-1)/(i-2) = i + 1
#
# clumsy(N):
#     if N = 1 => N
#     if N = 2 => N
#     if N = 3 => N + 3
#     if N = 4 => N + 2 + 1 = N + 3
#     if N > 4 and N % 4 == 1 => N + 1 + (... = 0) + 2 - 1           = N + 2
```

```python
#     if N > 4 and N % 4 == 2 => N + 1 + (... = 0) + 3 - 2 * 1       = N + 2
#     if N > 4 and N % 4 == 3 => N + 1 + (... = 0) + 4 - 3 * 2 / 1   = N - 1
#     if N > 4 and N % 4 == 0 => N + 1 + (... = 0) + 5 - (4*3/2) + 1 = N + 1

class Solution(object):
    def clumsy(self, N):
        """
        :type N: int
        :rtype: int
        """
        if N <= 2:
            return N
        if N <= 4:
            return N+3

        if N % 4 == 0:
            return N+1
        elif N % 4 <= 2:
            return N+2
        return N-1
```

next-greater-element-iii.py

```python
# Given a positive 32-bit integer n, you need to find the smallest 32-bit
# integer which has exactly the same digits existing in the integer n and is
# greater in value than n. If no such positive 32-bit integer exists, you need to
# return -1.
#
# Example 1:
#
# Input: 12
# Output: 21
#
#
#
#
# Example 2:
#
# Input: 21
# Output: -1# Time:  O(logn) = O(1)
# Space: O(logn) = O(1)


class Solution(object):
    def nextGreaterElement(self, n):
        """
        :type n: int
        :rtype: int
        """
        digits = map(int, list(str(n)))
        k, l = -1, 0
        for i in xrange(len(digits) - 1):
            if digits[i] < digits[i + 1]:
                k = i

        if k == -1:
            digits.reverse()
            return -1

        for i in xrange(k + 1, len(digits)):
            if digits[i] > digits[k]:
                l = i

        digits[k], digits[l] = digits[l], digits[k]
        digits[k + 1:] = digits[:k:-1]
        result = int("".join(map(str, digits)))
        return -1 if result >= 0x7FFFFFFF else result
```

permutations.py

```python
# Given a collection of distinct integers, return all possible permutations.
#
# Example:
#
# Input: [1,2,3]
# Output:
# [
#   [1,2,3],
#   [1,3,2],
#   [2,1,3],
#   [2,3,1],
#   [3,1,2],
#   [3,2,1]
# ]# Time:  O(n * n!)
# Space: O(n)

class Solution(object):
    # @param num, a list of integer
    # @return a list of lists of integers
    def permute(self, num):
        result = []
        used = [False] * len(num)
        self.permuteRecu(result, used, [], num)
        return result

    def permuteRecu(self, result, used, cur, num):
        if len(cur) == len(num):
            result.append(cur[:])
            return
        for i in xrange(len(num)):
            if not used[i]:
                used[i] = True
                cur.append(num[i])
                self.permuteRecu(result, used, cur, num)
                cur.pop()
                used[i] = False


# Time:  O(n^2 * n!)
# Space: O(n^2)
class Solution2(object):
    def permute(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        res = []
        self.dfs(nums, [], res)
        return res

    def dfs(self, nums, path, res):
        if not nums:
            res.append(path)

        for i in xrange(len(nums)):
            # e.g., [1, 2, 3]: 3! = 6 cases
            # idx -> nums, path
            # 0 -> [2, 3], [1] -> 0: [3], [1, 2] -> [], [1, 2, 3]
```

```python
#                  -> 1: [2], [1, 3] -> [], [1, 3, 2]
#
# 1 -> [1, 3], [2] -> 0: [3], [2, 1] -> [], [2, 1, 3]
#                  -> 1: [1], [2, 3] -> [], [2, 3, 1]
#
# 2 -> [1, 2], [3] -> 0: [2], [3, 1] -> [], [3, 1, 2]
#                  -> 1: [1], [3, 2] -> [], [3, 2, 1]
self.dfs(nums[:i] + nums[i+1:], path + [nums[i]], res)
```

number-of-longest-increasing-subsequence.py

```python
# Given an unsorted array of integers, find the number of longest increasing
# subsequence.
#
#
# Example 1:
#
# Input: [1,3,5,4,7]
# Output: 2
# Explanation: The two longest increasing subsequence are [1, 3, 4, 7] and [1,
# 3, 5, 7].
#
#
#
# Example 2:
#
# Input: [2,2,2,2,2]
# Output: 5
# Explanation: The length of longest continuous increasing subsequence is 1, and
# there are 5 subsequences' length is 1, so output 5.
#
#
#
# Note:
# Length of the given array will be not exceed 2000 and the answer is guaranteed
# to be fit in 32-bit signed int.# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def findNumberOfLIS(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result, max_len = 0, 0
        dp = [[1, 1] for _ in xrange(len(nums))]  # {length, number} pair
        for i in xrange(len(nums)):
            for j in xrange(i):
                if nums[i] > nums[j]:
                    if dp[i][0] == dp[j][0]+1:
                        dp[i][1] += dp[j][1]
                    elif dp[i][0] < dp[j][0]+1:
                        dp[i] = [dp[j][0]+1, dp[j][1]]
            if max_len == dp[i][0]:
                result += dp[i][1]
            elif max_len < dp[i][0]:
                max_len = dp[i][0]
                result = dp[i][1]
        return result
```

shortest-bridge.py

```
# In a given 2D binary array A, there are two islands.  (An island is a
# 4-directionally connected group of 1s not connected to any other 1s.)
#
# Now, we may change 0s to 1s so as to connect the two islands together to form
# 1 island.
#
# Return the smallest number of 0s that must be flipped.  (It is guaranteed that
# the answer is at least 1.)
#
#
# Example 1:
# Input: A = [[0,1],[1,0]]
# Output: 1
# Example 2:
# Input: A = [[0,1,0],[0,0,0],[0,0,1]]
# Output: 2
# Example 3:
# Input: A = [[1,1,1,1,1],[1,0,0,0,1],[1,0,1,0,1],[1,0,0,0,1],[1,1,1,1,1]]
# Output: 1
#
#
# Constraints:
#
#
#       2 <= A.length == A[0].length <= 100
#       A[i][j] == 0 or A[i][j] == 1# Time:  O(n^2)
# Space: O(n^2)

import collections


class Solution(object):
    def shortestBridge(self, A):
        """
        :type A: List[List[int]]
        :rtype: int
        """
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        def get_islands(A):
            islands = []
            done = set()
            for r, row in enumerate(A):
                for c, val in enumerate(row):
                    if val == 0 or (r, c) in done:
                        continue
                    s = [(r, c)]
                    lookup = set(s)
                    while s:
                        node = s.pop()
                        for d in directions:
                            nei = node[0]+d[0], node[1]+d[1]
                            if not (0 <= nei[0] < len(A) and 0 <= nei[1] < len(A[0])) or \
                               nei in lookup or A[nei[0]][nei[1]] == 0:
                                continue
                            s.append(nei)
                            lookup.add(nei)
                    done |= lookup
```

```
            islands.append(lookup)
            if len(islands) == 2:
                break
    return islands

lookup, target = get_islands(A)
q = collections.deque([(node, 0) for node in lookup])
while q:
    node, dis = q.popleft()
    if node in target:
        return dis-1
    for d in directions:
        nei = node[0]+d[0], node[1]+d[1]
        if not (0 <= nei[0] < len(A) and 0 <= nei[1] < len(A[0])) or \
           nei in lookup:
            continue
        q.append((nei, dis+1))
        lookup.add(nei)
```

complex-number-multiplication.py

```python
# Given two strings representing two complex numbers.
#
#
# You need to return a string representing their multiplication. Note i2 = -1
# according to the definition.
#
#
# Example 1:
#
# Input: "1+1i", "1+1i"
# Output: "0+2i"
# Explanation: (1 + i) * (1 + i) = 1 + i2 + 2 * i = 2i, and you need convert it
# to the form of 0+2i.
#
#
#
# Example 2:
#
# Input: "1+-1i", "1+-1i"
# Output: "0+-2i"
# Explanation: (1 - i) * (1 - i) = 1 + i2 - 2 * i = -2i, and you need convert it
# to the form of 0+-2i.
#
#
#
# Note:
#
# The input strings will not have extra blank.
# The input strings will be given in the form of a+bi, where the integer a and b
# will both belong to the range of [-100, 100]. And the output should be also in
# this form.# Time:  O(1)
# Space: O(1)

class Solution(object):
    def complexNumberMultiply(self, a, b):
        """
        :type a: str
        :type b: str
        :rtype: str
        """
        ra, ia = map(int, a[:-1].split('+'))
        rb, ib = map(int, b[:-1].split('+'))
        return '%d+%di' % (ra * rb - ia * ib, ra * ib + ia * rb)
```

copy-list-with-random-pointer.py

```python
# 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.
#
# The Linked List is represented in the input/output as a list of n nodes. Each
# node is represented as a pair of [val, random_index] where:
#
#
#       val: an integer representing Node.val
#       random_index: the index of the node (range from 0 to n-1) where random
# pointer points to, or null if it does not point to any node.
#
#
#
# Example 1:
#
# Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
# Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
#
#
# Example 2:
#
# Input: head = [[1,1],[2,1]]
# Output: [[1,1],[2,1]]
#
#
# Example 3:
#
#
#
# Input: head = [[3,null],[3,0],[3,null]]
# Output: [[3,null],[3,0],[3,null]]
#
#
# Example 4:
#
# Input: head = []
# Output: []
# Explanation: Given linked list is empty (null pointer), so return null.
#
#
#
# Constraints:
#
#
#       -10000 <= Node.val <= 10000
#       Node.random is null or pointing to a node in the linked list.
#       Number of Nodes will not exceed 1000.# Time:  O(n)
# Space: O(1)

class RandomListNode(object):
    def __init__(self, x):
        self.label = x
        self.next = None
        self.random = None

class Solution(object):
```

```python
    # @param head, a RandomListNode
    # @return a RandomListNode
    def copyRandomList(self, head):
        # copy and combine copied list with original list
        current = head
        while current:
            copied = RandomListNode(current.label)
            copied.next = current.next
            current.next = copied
            current = copied.next

        # update random node in copied list
        current = head
        while current:
            if current.random:
                current.next.random = current.random.next
            current = current.next.next

        # split copied list from combined one
        dummy = RandomListNode(0)
        copied_current, current = dummy, head
        while current:
            copied_current.next = current.next
            current.next = current.next.next
            copied_current, current = copied_current.next, current.next
        return dummy.next

# Time:  O(n)
# Space: O(n)
class Solution2(object):
    # @param head, a RandomListNode
    # @return a RandomListNode
    def copyRandomList(self, head):
        dummy = RandomListNode(0)
        current, prev, copies = head, dummy, {}

        while current:
            copied = RandomListNode(current.label)
            copies[current] = copied
            prev.next = copied
            prev, current = prev.next, current.next

        current = head
        while current:
            if current.random:
                copies[current].random = copies[current.random]
            current = current.next

        return dummy.next

# time: O(n)
# space: O(n)
from collections import defaultdict


class Solution3(object):
    def copyRandomList(self, head):
        """
        :type head: RandomListNode
        :rtype: RandomListNode
```

```python
"""
clone = defaultdict(lambda: RandomListNode(0))
clone[None] = None
cur = head

while cur:
    clone[cur].label = cur.label
    clone[cur].next = clone[cur.next]
    clone[cur].random = clone[cur.random]
    cur = cur.next

return clone[head]
```

container-with-most-water.py

```python
# 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 and n is at least 2.
#
#
#
#
#
# The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this
# case, the max area of water (blue section) the container can contain is 49.
#
#
#
# Example:
#
# Input: [1,8,6,2,5,4,8,3,7]
# Output: 49# Time:  O(n)
# Space: O(1)

class Solution(object):
    # @return an integer
    def maxArea(self, height):
        max_area, i, j = 0, 0, len(height) - 1
        while i < j:
            max_area = max(max_area, min(height[i], height[j]) * (j - i))
            if height[i] < height[j]:
                i += 1
            else:
                j -= 1
        return max_area
```

find-duplicate-file-in-system.py

```
# Given a list of directory info including directory path, and all the files
# with contents in this directory, you need to find out all the groups of
# duplicate files in the file system in terms of their paths.
#
# A group of duplicate files consists of at least two files that have exactly
# the same content.
#
# A single directory info string in the input list has the following format:
#
# "root/d1/d2/.../dm f1.txt(f1_content) f2.txt(f2_content) ...
# fn.txt(fn_content)"
#
# It means there are n files (f1.txt, f2.txt ... fn.txt with content f1_content,
# f2_content ... fn_content, respectively) in directory root/d1/d2/.../dm. Note
# that n >= 1 and m >= 0. If m = 0, it means the directory is just the root
# directory.
#
# The output is a list of group of duplicate file paths. For each group, it
# contains all the file paths of the files that have the same content. A file path
# is a string that has the following format:
#
# "directory_path/file_name.txt"
#
# Example 1:
#
# Input:
# ["root/a 1.txt(abcd) 2.txt(efgh)", "root/c 3.txt(abcd)", "root/c/d
# 4.txt(efgh)", "root 4.txt(efgh)"]
# Output:
#
# [["root/a/2.txt","root/c/d/4.txt","root/4.txt"],["root/a/1.txt","root/c/3.txt"]]
#
#
#
#
# Note:
#
#
#       No order is required for the final output.
#       You may assume the directory name, file name and file content only has
# letters and digits, and the length of file content is in the range of [1,50].
#       The number of files given is in the range of [1,20000].
#       You may assume no files or directories share the same name in the same
# directory.
#       You may assume each given directory info represents a unique directory.
# Directory path and file info are separated by a single blank space.
#
#
#
# Follow-up beyond contest:
#
#
#       Imagine you are given a real file system, how will you search files? DFS
# or BFS?
#       If the file content is very large (GB level), how will you modify your
# solution?
#       If you can only read the file by 1kb each time, how will you modify your
# solution?
```

```python
#       What is the time complexity of your modified solution? What is the most
# time-consuming part and memory consuming part of it? How to optimize?
#       How to make sure the duplicated files you find are not false positive?# Time:  O(n * l), l is the aver
# Space: O(n * l)

import collections


class Solution(object):
    def findDuplicate(self, paths):
        """
        :type paths: List[str]
        :rtype: List[List[str]]
        """
        files = collections.defaultdict(list)
        for path in paths:
           s = path.split(" ")
           for i in xrange(1,len(s)):
               file_name = s[0] + "/" + s[i][0:s[i].find("(")]
               file_content = s[i][s[i].find("(")+1:s[i].find(")")]
               files[file_content].append(file_name)

        result = []
        for file_content, file_names in files.iteritems():
            if len(file_names)>1:
                result.append(file_names)
        return result
```

fraction-to-recurring-decimal.py

```
# 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.
#
# If multiple answers are possible, just return any of them.
#
# Example 1:
#
# Input: numerator = 1, denominator = 2
# Output: "0.5"
#
#
# Example 2:
#
# Input: numerator = 2, denominator = 1
# Output: "2"
#
# Example 3:
#
# Input: numerator = 2, denominator = 3
# Output: "0.(6)"# Time:  O(logn), where logn is the length of result strings
# Space: O(1)

class Solution(object):
    def fractionToDecimal(self, numerator, denominator):
        """
        :type numerator: int
        :type denominator: int
        :rtype: str
        """
        result = ""
        if (numerator > 0 and denominator < 0) or (numerator < 0 and denominator > 0):
            result = "-"

        dvd, dvs = abs(numerator), abs(denominator)
        result += str(dvd / dvs)
        dvd %= dvs

        if dvd > 0:
            result += "."

        lookup = {}
        while dvd and dvd not in lookup:
            lookup[dvd] = len(result)
            dvd *= 10
            result += str(dvd / dvs)
            dvd %= dvs

        if dvd in lookup:
            result = result[:lookup[dvd]] + "(" + result[lookup[dvd]:] + ")"

        return result
```

path-with-maximum-probability.py

```python
# You are given an undirected weighted graph of n nodes (0-indexed), represented
# by an edge list where edges[i] = [a, b] is an undirected edge connecting the
# nodes a and b with a probability of success of traversing that edge succProb[i].
#
# Given two nodes start and end, find the path with the maximum probability of
# success to go from start to end and return its success probability.
#
# If there is no path from start to end, return 0. Your answer will be accepted
# if it differs from the correct answer by at most 1e-5.
#
#
# Example 1:
#
#
#
# Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.2], start =
# 0, end = 2
# Output: 0.25000
# Explanation: There are two paths from start to end, one having a probability
# of success = 0.2 and the other has 0.5 * 0.5 = 0.25.
#
#
# Example 2:
#
#
#
# Input: n = 3, edges = [[0,1],[1,2],[0,2]], succProb = [0.5,0.5,0.3], start =
# 0, end = 2
# Output: 0.30000
#
#
# Example 3:
#
#
#
# Input: n = 3, edges = [[0,1]], succProb = [0.5], start = 0, end = 2
# Output: 0.00000
# Explanation: There is no path between 0 and 2.
#
#
#
# Constraints:
#
#
#       2 <= n <= 10^4
#       0 <= start, end < n
#       start != end
#       0 <= a, b < n
#       a != b
#       0 <= succProb.length == edges.length <= 2*10^4
#       0 <= succProb[i] <= 1
#       There is at most one edge between every two nodes.# Time:  O((|E| + |V|) * log|V|) = O(|E| * log|V|) b
#        if we can further to use Fibonacci heap, it would be O(|E| + |V| * log|V|)
# Space: O(|E| + |V|) = O(|E|)

import collections
import itertools
import heapq
```

```python
class Solution(object):
    def maxProbability(self, n, edges, succProb, start, end):
        """
        :type n: int
        :type edges: List[List[int]]
        :type succProb: List[float]
        :type start: int
        :type end: int
        :rtype: float
        """
        adj = collections.defaultdict(list)
        for (u, v), p in itertools.izip(edges, succProb):
            adj[u].append((v, p))
            adj[v].append((u, p))
        max_heap = [(-1.0, start)]
        result, lookup = collections.defaultdict(float), set()
        result[start] = 1.0
        while max_heap and len(lookup) != len(adj):
            curr, u = heapq.heappop(max_heap)
            if u in lookup:
                continue
            lookup.add(u)
            for v, w in adj[u]:
                if v in lookup:
                    continue
                if v in result and result[v] >= -curr*w:
                    continue
                result[v] = -curr*w
                heapq.heappush(max_heap, (-result[v], v))
        return result[end]
```

house-robber-iii.py

```python
# 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:
#
# Input: [3,2,3,null,3,null,1]
#
#      3
#     / \
#    2   3
#     \   \
#      3   1
#
# Output: 7
# Explanation: Maximum amount of money the thief can rob = 3 + 3 + 1 = 7.
#
# Example 2:
#
# Input: [3,4,5,1,3,null,1]
#
#      3
#     / \
#    4   5
#   / \   \
#  1   3   1
#
# Output: 9
# Explanation: Maximum amount of money the thief can rob = 4 + 5 = 9.# Time:  O(n)
# Space: O(h)

class Solution(object):
    def rob(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def robHelper(root):
            if not root:
                return (0, 0)
            left, right = robHelper(root.left), robHelper(root.right)
            return (root.val + left[1] + right[1], max(left) + max(right))

        return max(robHelper(root))
```

letter-tile-possibilities.py

```python
# You have n  tiles, where each tile has one letter tiles[i] printed on it.
#
# Return the number of possible non-empty sequences of letters you can make
# using the letters printed on those tiles.
#
#
# Example 1:
#
# Input: tiles = "AAB"
# Output: 8
# Explanation: The possible sequences are "A", "B", "AA", "AB", "BA", "AAB",
# "ABA", "BAA".
#
#
# Example 2:
#
# Input: tiles = "AAABBC"
# Output: 188
#
#
# Example 3:
#
# Input: tiles = "V"
# Output: 1
#
#
#
# Constraints:
#
#
#       1 <= tiles.length <= 7
#       tiles consists of uppercase English letters.# Time:  O(n^2)
# Space: O(n)

import collections


class Solution(object):
    def numTilePossibilities(self, tiles):
        """
        :type tiles: str
        :rtype: int
        """
        fact = [0.0]*(len(tiles)+1)
        fact[0] = 1.0;
        for i in xrange(1, len(tiles)+1):
            fact[i] = fact[i-1]*i
        count = collections.Counter(tiles)

        # 1. we can represent each alphabet 1..26 as generating functions:
        #    G1(x) = 1 + x^1/1! + x^2/2! + x^3/3! + ... + x^count1/count1!
        #    G2(x) = 1 + x^1/1! + x^2/2! + x^3/3! + ... + x^count2/count2!
        #    ...
        #    G26(x) = 1 + x^1/1! + x^2/2! + x^3/3! + ... + x^count26/count26!
        #
        # 2. let G1(x)*G2(x)*...*G26(x) = c0 + c1*x1 + ... + ck*x^k, k is the max number s.t. ck != 0
        #    => ci (1 <= i <= k) is the number we need to divide when permuting i letters
        #    => the answer will be : c1*1! + c2*2! + ... + ck*k!
```

```python
        coeff = [0.0]*(len(tiles)+1)
        coeff[0] = 1.0
        for i in count.itervalues():
            new_coeff = [0.0]*(len(tiles)+1)
            for j in xrange(len(coeff)):
                for k in xrange(i+1):
                    if k+j >= len(new_coeff):
                        break
                    new_coeff[j+k] += coeff[j]*1.0/fact[k]
            coeff = new_coeff

        result = 0
        for i in xrange(1, len(coeff)):
            result += int(round(coeff[i]*fact[i]))
        return result


# Time:  O(r), r is the value of result
# Space: O(n)
class Solution2(object):
    def numTilePossibilities(self, tiles):
        """
        :type tiles: str
        :rtype: int
        """
        def backtracking(counter):
            total = 0
            for k, v in counter.iteritems():
                if not v:
                    continue
                counter[k] -= 1
                total += 1+backtracking(counter)
                counter[k] += 1
            return total

        return backtracking(collections.Counter(tiles))
```

sort-an-array.py

```
# Given an array of integers nums, sort the array in ascending order.
#
#
# Example 1:
# Input: nums = [5,2,3,1]
# Output: [1,2,3,5]
# Example 2:
# Input: nums = [5,1,1,2,0,0]
# Output: [0,0,1,1,2,5]
#
#
# Constraints:
#
#
#       1 <= nums.length <= 50000
#       -50000 <= nums[i] <= 50000# Time:  O(nlogn)
# Space: O(n)

# merge sort solution
class Solution(object):
    def sortArray(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        def mergeSort(start, end, nums):
            if end - start <= 1:
                return
            mid = start + (end - start) // 2
            mergeSort(start, mid, nums)
            mergeSort(mid, end,  nums)
            right = mid
            tmp = []
            for left in xrange(start, mid):
                while right < end and nums[right] < nums[left]:
                    tmp.append(nums[right])
                    right += 1
                tmp.append(nums[left])
            nums[start:start+len(tmp)] = tmp

        mergeSort(0, len(nums), nums)
        return nums


# Time:  O(nlogn), on average
# Space: O(logn)
import random
# quick sort solution
class Solution2(object):
    def sortArray(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        def kthElement(nums, left, k, right, compare):
            def PartitionAroundPivot(left, right, pivot_idx, nums, compare):
                new_pivot_idx = left
                nums[pivot_idx], nums[right] = nums[right], nums[pivot_idx]
```

```
        for i in xrange(left, right):
            if compare(nums[i], nums[right]):
                nums[i], nums[new_pivot_idx] = nums[new_pivot_idx], nums[i]
                new_pivot_idx += 1

        nums[right], nums[new_pivot_idx] = nums[new_pivot_idx], nums[right]
        return new_pivot_idx

    right -= 1
    while left <= right:
        pivot_idx = random.randint(left, right)
        new_pivot_idx = PartitionAroundPivot(left, right, pivot_idx, nums, compare)
        if new_pivot_idx == k:
            return
        elif new_pivot_idx > k:
            right = new_pivot_idx - 1
        else:  # new_pivot_idx < k.
            left = new_pivot_idx + 1

def quickSort(start, end, nums):
    if end - start <= 1:
        return
    mid = start + (end - start) / 2
    kthElement(nums, start, mid, end, lambda a, b: a < b)
    quickSort(start, mid, nums)
    quickSort(mid, end, nums)

quickSort(0, len(nums), nums)
return nums
```

the-dining-philosophers.py

```
# Five silent philosophers sit at a round table with bowls of spaghetti. Forks
# are placed between each pair of adjacent philosophers.
#
# Each philosopher must alternately think and eat. However, a philosopher can
# only eat spaghetti when they have both left and right forks. Each fork can be
# held by only one philosopher and so a philosopher can use the fork only if it is
# not being used by another philosopher. After an individual philosopher finishes
# eating, they need to put down both forks so that the forks become available to
# others. A philosopher can take the fork on their right or the one on their left
# as they become available, but cannot start eating before getting both forks.
#
# Eating is not limited by the remaining amounts of spaghetti or stomach space;
# an infinite supply and an infinite demand are assumed.
#
# Design a discipline of behaviour (a concurrent algorithm) such that no
# philosopher will starve; i.e., each can forever continue to alternate between
# eating and thinking, assuming that no philosopher can know when others may want
# to eat or think.
#
#
#
# The problem statement and the image above are taken from wikipedia.org
#
#
#
# The philosophers' ids are numbered from 0 to 4 in a clockwise order. Implement
# the function void wantsToEat(philosopher, pickLeftFork, pickRightFork, eat,
# putLeftFork, putRightFork) where:
#
#
#       philosopher is the id of the philosopher who wants to eat.
#       pickLeftFork and pickRightFork are functions you can call to pick the
# corresponding forks of that philosopher.
#       eat is a function you can call to let the philosopher eat once he has
# picked both forks.
#       putLeftFork and putRightFork are functions you can call to put down the
# corresponding forks of that philosopher.
#       The philosophers are assumed to be thinking as long as they are not
# asking to eat (the function is not being called with their number).
#
#
# Five threads, each representing a philosopher, will simultaneously use one
# object of your class to simulate the process. The function may be called for the
# same philosopher more than once, even before the last call ends.
#
#
# Example 1:
#
# Input: n = 1
# Output: [[4,2,1],[4,1,1],[0,1,1],[2,2,1],[2,1,1],[2,0,3],[2,1,2],[2,2,2],[4,0,
# 3],[4,1,2],[0,2,1],[4,2,2],[3,2,1],[3,1,1],[0,0,3],[0,1,2],[0,2,2],[1,2,1],[1,1,
# 1],[3,0,3],[3,1,2],[3,2,2],[1,0,3],[1,1,2],[1,2,2]]
# Explanation:
# n is the number of times each philosopher will call the function.
# The output array describes the calls you made to the functions controlling the
# forks and the eat function, its format is:
# output[i] = [a, b, c] (three integers)
# - a is the id of a philosopher.
```

```python
# - b specifies the fork: {1 : left, 2 : right}.
# - c specifies the operation: {1 : pick, 2 : put, 3 : eat}.
#
#
# Constraints:
#
#
#       1 <= n <= 60# Time:  O(n)
# Space: O(1)

import threading


class DiningPhilosophers(object):
    def __init__(self):
        self._l = [threading.Lock() for _ in xrange(5)]

    # call the functions directly to execute, for example, eat()
    def wantsToEat(self, philosopher, pickLeftFork, pickRightFork, eat, putLeftFork, putRightFork):
        """
        :type philosopher: int
        :type pickLeftFork: method
        :type pickRightFork: method
        :type eat: method
        :type putLeftFork: method
        :type putRightFork: method
        :rtype: void
        """
        left, right = philosopher, (philosopher+4)%5
        first, second = left, right
        if  philosopher%2 == 0:
            first, second = left, right
        else:
            first, second = right, left

        with self._l[first]:
            with self._l[second]:
                pickLeftFork()
                pickRightFork()
                eat()
                putLeftFork()
                putRightFork()
```

majority-element-ii.py

```python
# Given an integer array of size n, find all elements that appear more than
# n/3  times.
#
# Note: The algorithm should run in linear time and in O(1) space.
#
# Example 1:
#
# Input: [3,2,3]
# Output: [3]
#
# Example 2:
#
# Input: [1,1,1,3,3,2,2,2]
# Output: [1,2]# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def majorityElement(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        k, n, cnts = 3, len(nums), collections.defaultdict(int)

        for i in nums:
            cnts[i] += 1
            # Detecting k items in cnts, at least one of them must have exactly
            # one in it. We will discard those k items by one for each.
            # This action keeps the same mojority numbers in the remaining numbers.
            # Because if x / n  > 1 / k is true, then (x - 1) / (n - k) > 1 / k is also true.
            if len(cnts) == k:
                for j in cnts.keys():
                    cnts[j] -= 1
                    if cnts[j] == 0:
                        del cnts[j]

        # Resets cnts for the following counting.
        for i in cnts.keys():
            cnts[i] = 0

        # Counts the occurrence of each candidate integer.
        for i in nums:
            if i in cnts:
                cnts[i] += 1

        # Selects the integer which occurs > [n / k] times.
        result = []
        for i in cnts.keys():
            if cnts[i] > n / k:
                result.append(i)

        return result

    def majorityElement2(self, nums):
        """
```

```
    :type nums: List[int]
    :rtype: List[int]
    """
    return [i[0] for i in collections.Counter(nums).items() if i[1] > len(nums) / 3]
```

reordered-power-of-2.py

```python
# Example 4:
#
# Input: 24
# Output: false
#
#
#
# Example 5:
#
# Input: 46
# Output: true
#
#
#
#
# Note:
#
#
#       1 <= N <= 10^9# Time:  O((logn)^2) = O(1) due to n is a 32-bit number
# Space: O(logn) = O(1)

import collections


class Solution(object):
    def reorderedPowerOf2(self, N):
        """
        :type N: int
        :rtype: bool
        """
        count = collections.Counter(str(N))
        return any(count == collections.Counter(str(1 << i))
                   for i in xrange(31))
```

smallest-string-starting-from-leaf.py

```python
# Example 1:
#
#
#
# Input: [0,1,2,3,4,3,4]
# Output: "dba"
#
#
#
# Example 2:
#
#
#
# Input: [25,1,3,1,3,0,2]
# Output: "adz"
#
#
#
# Example 3:
#
#
#
# Input: [2,2,1,null,1,0,null,0]
# Output: "abc"
#
#
#
#
# Note:
#
#
#       The number of nodes in the given tree will be between 1 and 8500.
#       Each node in the tree will have a value between 0 and 25.# Time:  O(n + l * h), l is the number of lea
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def smallestFromLeaf(self, root):
        """
        :type root: TreeNode
        :rtype: str
        """
        def dfs(node, candidate, result):
            if not node:
                return

            candidate.append(chr(ord('a') + node.val))
            if not node.left and not node.right:
                result[0] = min(result[0], "".join(reversed(candidate)))
            dfs(node.left, candidate, result)
            dfs(node.right, candidate, result)
```

```
        candidate.pop()

    result = ["~"]
    dfs(root, [], result)
    return result[0]
```

delete-nodes-and-return-forest.py

```python
# Given the root of a binary tree, each node in the tree has a distinct value.
#
# After deleting all nodes with a value in to_delete, we are left with a forest
# (a disjoint union of trees).
#
# Return the roots of the trees in the remaining forest.  You may return the
# result in any order.
#
#
# Example 1:
#
#
#
# Input: root = [1,2,3,4,5,6,7], to_delete = [3,5]
# Output: [[1,2,null,4],[6],[7]]
#
#
#
# Constraints:
#
#
#       The number of nodes in the given tree is at most 1000.
#       Each node has a distinct value between 1 and 1000.
#       to_delete.length <= 1000
#       to_delete contains distinct values between 1 and 1000.# Time:  O(n)
# Space: O(h + d), d is the number of to_delete

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def delNodes(self, root, to_delete):
        """
        :type root: TreeNode
        :type to_delete: List[int]
        :rtype: List[TreeNode]
        """
        def delNodesHelper(to_delete_set, root, is_root, result):
            if not root:
                return None
            is_deleted = root.val in to_delete_set
            if is_root and not is_deleted:
                result.append(root)
            root.left = delNodesHelper(to_delete_set, root.left, is_deleted, result)
            root.right = delNodesHelper(to_delete_set, root.right, is_deleted, result)
            return None if is_deleted else root

        result = []
        to_delete_set = set(to_delete)
        delNodesHelper(to_delete_set, root, True, result)
        return result
```

flip-columns-for-maximum-number-of-equal-rows.py

```python
# Example 2:
#
# Input: [[0,1],[1,0]]
# Output: 2
# Explanation: After flipping values in the first column, both rows have equal
# values.
#
#
#
# Example 3:
#
# Input: [[0,0,0],[0,0,1],[1,1,0]]
# Output: 2
# Explanation: After flipping values in the first two columns, the last two rows
# have equal values.
#
#
#
#
# Note:
#
#
#       1 <= matrix.length <= 300
#       1 <= matrix[i].length <= 300
#       All matrix[i].length's are equal
#       matrix[i][j] is 0 or 1# Time:  O(m * n)
# Space: O(m * n)

import collections


class Solution(object):
    def maxEqualRowsAfterFlips(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: int
        """
        count = collections.Counter(tuple(x^row[0] for x in row)
                                          for row in matrix)
        return max(count.itervalues())
```

advantage-shuffle.py

```python
# Example 2:
#
# Input: A = [12,24,8,32], B = [13,25,32,11]
# Output: [24,32,8,12]
#
#
#
#
# Note:
#
#
#       1 <= A.length = B.length <= 10000
#       0 <= A[i] <= 10^9
#       0 <= B[i] <= 10^9# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def advantageCount(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
        :rtype: List[int]
        """
        sortedA = sorted(A)
        sortedB = sorted(B)

        candidates = {b: [] for b in B}
        others = []
        j = 0
        for a in sortedA:
            if a > sortedB[j]:
                candidates[sortedB[j]].append(a)
                j += 1
            else:
                others.append(a)
        return [candidates[b].pop() if candidates[b] else others.pop()
                for b in B]
```

string-without-aaa-or-bbb.py

```python
# Given two integers A and B, return any string S such that:
#
#
#       S has length A + B and contains exactly A 'a' letters, and exactly B 'b'
# letters;
#       The substring 'aaa' does not occur in S;
#       The substring 'bbb' does not occur in S.
#
#
#
#
# Example 1:
#
# Input: A = 1, B = 2
# Output: "abb"
# Explanation: "abb", "bab" and "bba" are all correct answers.
#
#
#
# Example 2:
#
# Input: A = 4, B = 1
# Output: "aabaa"
#
#
#
#
# Note:
#
#
#       0 <= A <= 100
#       0 <= B <= 100
#       It is guaranteed such an S exists for the given A and B.# Time:  O(a + b)
# Space: O(1)

class Solution(object):
    def strWithout3a3b(self, A, B):
        """
        :type A: int
        :type B: int
        :rtype: str
        """
        result = []
        put_A = None
        while A or B:
            if len(result) >= 2 and result[-1] == result[-2]:
                put_A = result[-1] == 'b'
            else:
                put_A = A >= B

            if put_A:
                A -= 1
                result.append('a')
            else:
                B -= 1
                result.append('b')
        return "".join(result)
```

populating-next-right-pointers-in-each-node.py

```python
# You are given a perfect binary tree where all leaves are on the same level,
# and every parent has two children. The binary tree has the following definition:
#
# struct Node {
#   int val;
#   Node *left;
#   Node *right;
#   Node *next;
# }
#
#
# 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.
#
#
#
# Follow up:
#
#
#       You may only use constant extra space.
#       Recursive approach is fine, you may assume implicit stack space does not
# count as extra space for this problem.
#
#
#
# Example 1:
#
#
#
# Input: root = [1,2,3,4,5,6,7]
# Output: [1,#,2,3,#,4,5,6,7,#]
# Explanation: Given the above perfect binary tree (Figure A), your function
# should populate each next pointer to point to its next right node, just like in
# Figure B. The serialized output is in level order as connected by the next
# pointers, with '#' signifying the end of each level.
#
#
#
# Constraints:
#
#
#       The number of nodes in the given tree is less than 4096.
#       -1000 <= node.val <= 1000# Time:  O(n)
# Space: O(1)

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
        self.next = None

    def __repr__(self):
        if self is None:
            return "Nil"
        else:
```

```python
            return "{} -> {}".format(self.val, repr(self.next))

class Solution(object):
    # @param root, a tree node
    # @return nothing
    def connect(self, root):
        head = root
        while head:
            cur = head
            while cur and cur.left:
                cur.left.next = cur.right
                if cur.next:
                    cur.right.next = cur.next.left
                cur = cur.next
            head = head.left

# Time:  O(n)
# Space: O(logn)
# recusion
class Solution2(object):
    # @param root, a tree node
    # @return nothing
    def connect(self, root):
        if root is None:
            return
        if root.left:
            root.left.next = root.right
        if root.right and root.next:
            root.right.next = root.next.left
        self.connect(root.left)
        self.connect(root.right)
```

sum-of-nodes-with-even-valued-grandparent.py

```python
# Given a binary tree, return the sum of values of nodes with even-valued
# grandparent.  (A grandparent of a node is the parent of its parent, if it
# exists.)
#
# If there are no nodes with an even-valued grandparent, return 0.
#
#
# Example 1:
#
#
#
# Input: root = [6,7,8,2,7,1,3,9,null,1,4,null,null,null,5]
# Output: 18
# Explanation: The red nodes are the nodes with even-value grandparent while the
# blue nodes are the even-value grandparents.
#
#
#
# Constraints:
#
#
#       The number of nodes in the tree is between 1 and 10^4.
#       The value of nodes is between 1 and 100.# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def sumEvenGrandparent(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def sumEvenGrandparentHelper(root, p, gp):
            return sumEvenGrandparentHelper(root.left, root.val, p) + \
                   sumEvenGrandparentHelper(root.right, root.val, p) + \
                   (root.val if gp is not None and gp % 2 == 0 else 0) if root else 0

        return sumEvenGrandparentHelper(root, None, None)
```

print-zero-even-odd.py

```python
# Suppose you are given the following code:
#
# class ZeroEvenOdd {
#   public ZeroEvenOdd(int n) { ... }      // constructor
#   public void zero(printNumber) { ... }  // only output 0's
#   public void even(printNumber) { ... }  // only output even numbers
#   public void odd(printNumber) { ... }   // only output odd numbers
# }
#
#
# The same instance of ZeroEvenOdd will be passed to three different threads:
#
#
#       Thread A will call zero() which should only output 0's.
#       Thread B will call even() which should only ouput even numbers.
#       Thread C will call odd() which should only output odd numbers.
#
#
# Each of the threads is given a printNumber method to output an integer. Modify
# the given program to output the series 010203040506... where the length of the
# series must be 2n.
#
#
#
# Example 1:
#
# Input: n = 2
# Output: "0102"
# Explanation: There are three threads being fired asynchronously. One of them
# calls zero(), the other calls even(), and the last one calls odd(). "0102" is
# the correct output.
#
#
# Example 2:
#
# Input: n = 5
# Output: "0102030405"# Time:  O(n)
# Space: O(1)

import threading


class ZeroEvenOdd(object):
    def __init__(self, n):
        self.__n = n
        self.__curr = 0
        self.__cv = threading.Condition()

    # printNumber(x) outputs "x", where x is an integer.
    def zero(self, printNumber):
        """
        :type printNumber: method
        :rtype: void
        """
        for i in xrange(self.__n):
            with self.__cv:
                while self.__curr % 2 != 0:
                    self.__cv.wait()
```

```python
            self.__curr += 1
            printNumber(0)
            self.__cv.notifyAll()

    def even(self, printNumber):
        """
        :type printNumber: method
        :rtype: void
        """
        for i in xrange(2, self.__n+1, 2):
            with self.__cv:
                while self.__curr % 4 != 3:
                    self.__cv.wait()
                self.__curr += 1
                printNumber(i)
                self.__cv.notifyAll()

    def odd(self, printNumber):
        """
        :type printNumber: method
        :rtype: void
        """
        for i in xrange(1, self.__n+1, 2):
            with self.__cv:
                while self.__curr % 4 != 1:
                    self.__cv.wait()
                self.__curr += 1
                printNumber(i)
                self.__cv.notifyAll()
```

matchsticks-to-square.py

```
# Remember the story of Little Match Girl? By now, you know exactly what
# matchsticks the little match girl has, please find out a way you can make one
# square by using up all those matchsticks. You should not break any stick, but
# you can link them up, and each matchstick must be used exactly one time.
#
#  Your input will be several matchsticks the girl has, represented with their
# stick length. Your output will either be true or false, to represent whether you
# could make one square using all the matchsticks the little match girl has.
#
# Example 1:
#
# Input: [1,1,2,2,2]
# Output: true
#
# Explanation: You can form a square with length 2, one side of the square came
# two sticks with length 1.
#
#
#
# Example 2:
#
# Input: [3,3,3,3,4]
# Output: false
#
# Explanation: You cannot find a way to form a square with all the matchsticks.
#
#
#
# Note:
#
#
# The length sum of the given matchsticks is in the range of 0 to 10^9.
# The length of the given matchstick array will not exceed 15.# Time:  O(n * s * 2^n), s is the number of subs
# Space: O(n * (2^n + s))

class Solution(object):
    def makesquare(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        total_len = sum(nums)
        if total_len % 4:
            return False

        side_len = total_len / 4
        fullset = (1 << len(nums)) - 1

        used_subsets = []
        valid_half_subsets = [0] * (1 << len(nums))

        for subset in xrange(fullset+1):
            subset_total_len = 0
            for i in xrange(len(nums)):
                if subset & (1 << i):
                    subset_total_len += nums[i]

            if subset_total_len == side_len:
```

```python
        for used_subset in used_subsets:
            if (used_subset & subset) == 0:
                valid_half_subset = used_subset | subset
                valid_half_subsets[valid_half_subset] = True
                if valid_half_subsets[fullset ^ valid_half_subset]:
                    return True
        used_subsets.append(subset)

return False
```

remove-zero-sum-consecutive-nodes-from-linked-list.py

```python
# Given the head of a linked list, we repeatedly delete consecutive sequences of
# nodes that sum to 0 until there are no such sequences.
#
# After doing so, return the head of the final linked list.  You may return any
# such answer.
#
#
# (Note that in the examples below, all sequences are serializations of ListNode
# objects.)
#
# Example 1:
#
# Input: head = [1,2,-3,3,1]
# Output: [3,1]
# Note: The answer [1,2,1] would also be accepted.
#
#
# Example 2:
#
# Input: head = [1,2,3,-3,4]
# Output: [1,2,4]
#
#
# Example 3:
#
# Input: head = [1,2,3,-3,-2]
# Output: [1]
#
#
#
# Constraints:
#
#
#       The given linked list will contain between 1 and 1000 nodes.
#       Each node in the linked list has -1000 <= node.val <= 1000.# Time:  O(n)
# Space: O(n)

import collections


# Definition for singly-linked list.
class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None


class Solution(object):
    def removeZeroSumSublists(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        curr = dummy = ListNode(0)
        dummy.next = head
        prefix = 0
        lookup = collections.OrderedDict()
        while curr:
```

```python
        prefix += curr.val
        node = lookup.get(prefix, curr)
        while prefix in lookup:
            lookup.popitem()
        lookup[prefix] = node
        node.next = curr.next
        curr = curr.next
    return dummy.next
```

sort-list.py

```python
# Sort a linked list in O(n log n) time using constant space complexity.
#
# Example 1:
#
# Input: 4->2->1->3
# Output: 1->2->3->4
#
#
# Example 2:
#
# Input: -1->5->3->4->0
# Output: -1->0->3->4->5# Time:  O(nlogn)
# Space: O(logn) for stack call

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

    def __repr__(self):
        if self:
            return "{} -> {}".format(self.val, repr(self.next))

class Solution(object):
    # @param head, a ListNode
    # @return a ListNode
    def sortList(self, head):
        if head == None or head.next == None:
            return head

        fast, slow, prev = head, head, None
        while fast != None and fast.next != None:
            prev, fast, slow = slow, fast.next.next, slow.next
        prev.next = None

        sorted_l1 = self.sortList(head)
        sorted_l2 = self.sortList(slow)

        return self.mergeTwoLists(sorted_l1, sorted_l2)

    def mergeTwoLists(self, l1, l2):
        dummy = ListNode(0)

        cur = dummy
        while l1 != None and l2 != None:
            if l1.val <= l2.val:
                cur.next, cur, l1 = l1, l1, l1.next
            else:
                cur.next, cur, l2 = l2, l2, l2.next

        if l1 != None:
            cur.next = l1
        if l2 != None:
            cur.next = l2

        return dummy.next
```

guess-number-higher-or-lower-ii.py

```
# We are playing the Guess Game. The game is as follows:
#
# I pick a number from 1 to n. You have to guess which number I picked.
#
# Every time you guess wrong, I'll tell you whether the number I picked is
# higher or lower.
#
# However, when you guess a particular number x, and you guess wrong, you pay
# $x. You win the game when you guess the number I picked.
#
# Example:
#
# n = 10, I pick 8.
#
# First round:  You guess 5, I tell you that it's higher. You pay $5.
# Second round: You guess 7, I tell you that it's higher. You pay $7.
# Third round:  You guess 9, I tell you that it's lower. You pay $9.
#
# Game over. 8 is the number I picked.
#
# You end up paying $5 + $7 + $9 = $21.
#
#
# Given a particular n  1, find out how much money you need to have to
# guarantee a win.# Time:  O(n^2)
# Space: O(n^2)

class Solution(object):
    def getMoneyAmount(self, n):
        """
        :type n: int
        :rtype: int
        """
        pay = [[0] * n for _ in xrange(n+1)]
        for i in reversed(xrange(n)):
            for j in xrange(i+1, n):
                pay[i][j] = min(k+1 + max(pay[i][k-1], pay[k+1][j]) \
                                for k in xrange(i, j+1))
        return pay[0][n-1]
```

flatten-nested-list-iterator.py

```python
# 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:
#
#
# Input: [[1,1],2,[1,1]]
# Output: [1,1,2,1,1]
# Explanation: By calling next repeatedly until hasNext returns false,
#              the order of elements returned by next should be: [1,1,2,1,1].
#
#
# Example 2:
#
# Input: [1,[4,[6]]]
# Output: [1,4,6]
# Explanation: By calling next repeatedly until hasNext returns false,
#              the order of elements returned by next should be: [1,4,6].# Time:  O(n), n is the number of the
# Space: O(h), h is the depth of the nested lists.

class NestedIterator(object):

    def __init__(self, nestedList):
        """
        Initialize your data structure here.
        :type nestedList: List[NestedInteger]
        """
        self.__depth = [[nestedList, 0]]


    def next(self):
        """
        :rtype: int
        """
        nestedList, i = self.__depth[-1]
        self.__depth[-1][1] += 1
        return nestedList[i].getInteger()


    def hasNext(self):
        """
        :rtype: bool
        """
        while self.__depth:
            nestedList, i = self.__depth[-1]
            if i == len(nestedList):
                self.__depth.pop()
            elif nestedList[i].isInteger():
                    return True
            else:
                self.__depth[-1][1] += 1
                self.__depth.append([nestedList[i].getList(), 0])
        return False
```

flatten-binary-tree-to-linked-list.py

```python
# Given a binary tree, flatten it to a linked list in-place.
#
# For example, given the following tree:
#
#      1
#     / \
#    2   5
#   / \   \
# 3   4   6
#
#
# The flattened tree should look like:
#
# 1
#   \
#    2
#     \
#      3
#       \
#        4
#         \
#          5
#           \
#            6# Time:  O(n)
# Space: O(h), h is height of binary tree

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    # @param root, a tree node
    # @return nothing, do it in place
    def flatten(self, root):
        self.flattenRecu(root, None)

    def flattenRecu(self, root, list_head):
        if root:
            list_head = self.flattenRecu(root.right, list_head)
            list_head = self.flattenRecu(root.left, list_head)
            root.right = list_head
            root.left = None
            return root
        else:
            return list_head

class Solution2(object):
    list_head = None
    # @param root, a tree node
    # @return nothing, do it in place
    def flatten(self, root):
        if root:
            self.flatten(root.right)
            self.flatten(root.left)
            root.right = self.list_head
            root.left = None
```

```
self.list_head = root
```

h-index-ii.py

```python
# Given an array of citations sorted in ascending order (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."
#
# Example:
#
# Input: citations = [0,1,3,5,6]
# Output: 3
# Explanation: [0,1,3,5,6] means the researcher has 5 papers in total and each
# of them had
#              received 0, 1, 3, 5, 6 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, her h-index is 3.
#
# Note:
#
# If there are several possible values for h, the maximum one is taken as the
# h-index.
#
# Follow up:
#
#
#       This is a follow up problem to H-Index, where citations is now
# guaranteed to be sorted in ascending order.
#       Could you solve it in logarithmic time complexity?# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def hIndex(self, citations):
        """
        :type citations: List[int]
        :rtype: int
        """
        n = len(citations)
        left, right = 0, n - 1
        while left <= right:
            mid = (left + right) / 2
            if citations[mid] >= n - mid:
                right = mid - 1
            else:
                left = mid + 1
        return n - left
```

path-sum-ii.py

```python
# Given a binary tree and a sum, find all root-to-leaf paths where each path's
# sum equals the given sum.
#
# Note: A leaf is a node with no children.
#
# Example:
#
# Given the below binary tree and sum = 22,
#
#       5
#      / \
#     4   8
#    /   / \
#   11  13  4
#  /  \    / \
# 7    2  5   1
#
#
# Return:
#
# [
#    [5,4,11,2],
#    [5,8,4,5]
# ]# Time:  O(n)
# Space: O(h), h is height of binary tree

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    # @param root, a tree node
    # @param sum, an integer
    # @return a list of lists of integers
    def pathSum(self, root, sum):
        return self.pathSumRecu([], [], root, sum)


    def pathSumRecu(self, result, cur, root, sum):
        if root is None:
            return result

        if root.left is None and root.right is None and root.val == sum:
            result.append(cur + [root.val])
            return result

        cur.append(root.val)
        self.pathSumRecu(result, cur, root.left, sum - root.val)
        self.pathSumRecu(result, cur,root.right, sum - root.val)
        cur.pop()
        return result
```

beautiful-arrangement.py

```
# Suppose you have N integers from 1 to N. We define a beautiful arrangement as
# an array that is constructed by these N numbers successfully if one of the
# following is true for the ith position (1 <= i <= N) in this array:
#
#
#       The number at the ith position is divisible by i.
#       i is divisible by the number at the ith position.
#
#
#
#
# Now given N, how many beautiful arrangements can you construct?
#
# Example 1:
#
# Input: 2
# Output: 2
# Explanation:
#
# The first beautiful arrangement is [1, 2]:
#
# Number at the 1st position (i=1) is 1, and 1 is divisible by i (i=1).
#
# Number at the 2nd position (i=2) is 2, and 2 is divisible by i (i=2).
#
# The second beautiful arrangement is [2, 1]:
#
# Number at the 1st position (i=1) is 2, and 2 is divisible by i (i=1).
#
# Number at the 2nd position (i=2) is 1, and i (i=2) is divisible by 1.
#
#
#
#
# Note:
#
#
#       N is a positive integer and will not exceed 15.# Time:  O(n!)
# Space: O(n)


class Solution(object):
    def countArrangement(self, N):
        """
        :type N: int
        :rtype: int
        """
        def countArrangementHelper(n, arr):
            if n <= 0:
                return 1
            count = 0
            for i in xrange(n):
                if arr[i] % n == 0 or n % arr[i] == 0:
                    arr[i], arr[n-1] = arr[n-1], arr[i]
                    count += countArrangementHelper(n - 1, arr)
                    arr[i], arr[n-1] = arr[n-1], arr[i]
            return count
```

```
    return countArrangementHelper(N, range(1, N+1))
```

rotate-function.py

```python
# Given an array of integers A and let n to be its length.
#
#
#
# Assume Bk to be an array obtained by rotating the array A k positions clock-
# wise, we define a "rotation function" F on A as follow:
#
#
#
# F(k) = 0 * Bk[0] + 1 * Bk[1] + ... + (n-1) * Bk[n-1].
#
# Calculate the maximum value of F(0), F(1), ..., F(n-1).
#
#
# Note:
#
# n is guaranteed to be less than 105.
#
#
# Example:
# A = [4, 3, 2, 6]
#
# F(0) = (0 * 4) + (1 * 3) + (2 * 2) + (3 * 6) = 0 + 3 + 4 + 18 = 25
# F(1) = (0 * 6) + (1 * 4) + (2 * 3) + (3 * 2) = 0 + 4 + 6 + 6 = 16
# F(2) = (0 * 2) + (1 * 6) + (2 * 4) + (3 * 3) = 0 + 6 + 8 + 9 = 23
# F(3) = (0 * 3) + (1 * 2) + (2 * 6) + (3 * 4) = 0 + 2 + 12 + 12 = 26
#
# So the maximum value of F(0), F(1), F(2), F(3) is F(3) = 26.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def maxRotateFunction(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        s = sum(A)
        fi = 0
        for i in xrange(len(A)):
            fi += i * A[i]

        result = fi
        for i in xrange(1, len(A)+1):
            fi += s - len(A) * A[-i]
            result = max(result, fi)
        return result
```

partition-array-for-maximum-sum.py

```python
# Given an integer array A, you partition the array into (contiguous) subarrays
# of length at most K.  After partitioning, each subarray has their values changed
# to become the maximum value of that subarray.
#
# Return the largest sum of the given array after partitioning.
#
#
#
# Example 1:
#
# Input: A = [1,15,7,9,2,5,10], K = 3
# Output: 84
# Explanation: A becomes [15,15,15,9,10,10,10]
#
#
#
# Note:
#
#
#       1 <= K <= A.length <= 500
#       0 <= A[i] <= 10^6# Time:  O(n * k)
# Space: O(k)

class Solution(object):
    def maxSumAfterPartitioning(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        W = K+1
        dp = [0]*W
        for i in xrange(len(A)):
            curr_max = 0
            # dp[i % W] = 0;  # no need in this problem
            for k in xrange(1, min(K, i+1) + 1):
                curr_max = max(curr_max, A[i-k+1])
                dp[i % W] = max(dp[i % W], (dp[(i-k) % W] if i >= k else 0) + curr_max*k)
        return dp[(len(A)-1) % W]
```

shopping-offers.py

```
# In LeetCode Store, there are some kinds of items to sell. Each item has a
# price.
#
#
#
# However, there are some special offers, and a special offer consists of one or
# more different kinds of items with a sale price.
#
#
#
# You are given the each item's price, a set of special offers, and the number
# we need to buy for each item.
# The job is to output the lowest price you have to pay for exactly certain
# items as given, where you could make optimal use of the special offers.
#
#
#
# Each special offer is represented in the form of an array, the last number
# represents the price you need to pay for this special offer, other numbers
# represents how many specific items you could get if you buy this offer.
#
#
# You could use any of special offers as many times as you want.
#
# Example 1:
#
# Input: [2,5], [[3,0,5],[1,2,10]], [3,2]
# Output: 14
# Explanation:
# There are two kinds of items, A and B. Their prices are $2 and $5
# respectively.
# In special offer 1, you can pay $5 for 3A and 0B
# In special offer 2, you can pay $10 for 1A and 2B.
# You need to buy 3A and 2B, so you may pay $10 for 1A and 2B (special offer
# #2), and $4 for 2A.
#
#
#
# Example 2:
#
# Input: [2,3,4], [[1,1,0,4],[2,2,1,9]], [1,2,1]
# Output: 11
# Explanation:
# The price of A is $2, and $3 for B, $4 for C.
# You may pay $4 for 1A and 1B, and $9 for 2A ,2B and 1C.
# You need to buy 1A ,2B and 1C, so you may pay $4 for 1A and 1B (special offer
# #1), and $3 for 1B, $4 for 1C.
# You cannot add more items, though only $9 for 2A ,2B and 1C.
#
#
#
# Note:
#
#
# There are at most 6 kinds of items, 100 special offers.
# For each item, you need to buy at most 6 of them.
# You are not allowed to buy more items than you want, even if that would lower
# the overall price.# Time:  O(n * 2^n)
```

```python
# Space: O(n)

class Solution(object):
    def shoppingOffers(self, price, special, needs):
        """
        :type price: List[int]
        :type special: List[List[int]]
        :type needs: List[int]
        :rtype: int
        """
        def shoppingOffersHelper(price, special, needs, i):
            if i == len(special):
                return sum(map(lambda x, y: x*y, price, needs))
            result = shoppingOffersHelper(price, special, needs, i+1)
            for j in xrange(len(needs)):
                needs[j] -= special[i][j]
            if all(need >= 0 for need in needs):
                result = min(result, special[i][-1] + shoppingOffersHelper(price, special, needs, i))
            for j in xrange(len(needs)):
                needs[j] += special[i][j]
            return result

        return shoppingOffersHelper(price, special, needs, 0)
```

construct-binary-tree-from-inorder-and-postorder-traversal.py

```
# Given inorder and postorder traversal of a tree, construct the binary tree.
#
# Note:
#
# You may assume that duplicates do not exist in the tree.
#
# For example, given
#
# inorder = [9,3,15,20,7]
# postorder = [9,15,7,20,3]
#
# Return the following binary tree:
#
#     3
#    / \
#   9  20
#     /  \
#    15   7# Time:  O(n)
# Space: O(n)

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
    # @param inorder, a list of integers
    # @param postorder, a list of integers
    # @return a tree node
    def buildTree(self, inorder, postorder):
        lookup = {}
        for i, num in enumerate(inorder):
            lookup[num] = i
        return self.buildTreeRecu(lookup, postorder, inorder, len(postorder), 0, len(inorder))

    def buildTreeRecu(self, lookup, postorder, inorder, post_end, in_start, in_end):
        if in_start == in_end:
            return None
        node = TreeNode(postorder[post_end - 1])
        i = lookup[postorder[post_end - 1]]
        node.left = self.buildTreeRecu(lookup, postorder, inorder, post_end - 1 - (in_end - i - 1), in_start,
        node.right = self.buildTreeRecu(lookup, postorder, inorder, post_end - 1, i + 1, in_end)
        return node
```

teemo-attacking.py

```python
# In LOL world, there is a hero called Teemo and his attacking can make his
# enemy Ashe be in poisoned condition. Now, given the Teemo's attacking ascending
# time series towards Ashe and the poisoning time duration per Teemo's attacking,
# you need to output the total time that Ashe is in poisoned condition.
#
# You may assume that Teemo attacks at the very beginning of a specific time
# point, and makes Ashe be in poisoned condition immediately.
#
# Example 1:
#
# Input: [1,4], 2
# Output: 4
# Explanation: At time point 1, Teemo starts attacking Ashe and makes Ashe be
# poisoned immediately.
# This poisoned status will last 2 seconds until the end of time point 2.
# And at time point 4, Teemo attacks Ashe again, and causes Ashe to be in
# poisoned status for another 2 seconds.
# So you finally need to output 4.
#
#
#
#
# Example 2:
#
# Input: [1,2], 2
# Output: 3
# Explanation: At time point 1, Teemo starts attacking Ashe and makes Ashe be
# poisoned.
# This poisoned status will last 2 seconds until the end of time point 2.
# However, at the beginning of time point 2, Teemo attacks Ashe again who is
# already in poisoned status.
# Since the poisoned status won't add up together, though the second poisoning
# attack will still work at time point 2, it will stop at the end of time point 3.
# So you finally need to output 3.
#
#
#
#
# Note:
#
#
#       You may assume the length of given time series array won't exceed 10000.
#       You may assume the numbers in the Teemo's attacking time series and his
# poisoning time duration per attacking are non-negative integers, which won't
# exceed 10,000,000.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findPoisonedDuration(self, timeSeries, duration):
        """
        :type timeSeries: List[int]
        :type duration: int
        :rtype: int
        """
        result = duration * len(timeSeries)
        for i in xrange(1, len(timeSeries)):
            result -= max(0, duration - (timeSeries[i] - timeSeries[i-1]))
        return result
```

largest-divisible-subset.py

```python
# Input: [1,2,3]
# Output: [1,2] (of course, [1,3] will also be ok)
#
#
#
# Example 2:
#
# Input: [1,2,4,8]
# Output: [1,2,4,8]# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def largestDivisibleSubset(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        if not nums:
            return []

        nums.sort()
        dp = [1] * len(nums)
        prev = [-1] * len(nums)
        largest_idx = 0
        for i in xrange(len(nums)):
            for j in xrange(i):
                if nums[i] % nums[j] == 0:
                    if dp[i] < dp[j] + 1:
                        dp[i] = dp[j] + 1
                        prev[i] = j
            if dp[largest_idx] < dp[i]:
                largest_idx = i

        result = []
        i = largest_idx
        while i != -1:
            result.append(nums[i])
            i = prev[i]
        return result[::-1]
```

word-subsets.py

```python
# Example 1:
#
# Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["e","o"]
# Output: ["facebook","google","leetcode"]
#
#
#
# Example 2:
#
# Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["l","e"]
# Output: ["apple","google","leetcode"]
#
#
#
# Example 3:
#
# Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["e","oo"]
# Output: ["facebook","google"]
#
#
#
# Example 4:
#
# Input: A = ["amazon","apple","facebook","google","leetcode"], B = ["lo","eo"]
# Output: ["google","leetcode"]
#
#
#
# Example 5:
#
# Input: A = ["amazon","apple","facebook","google","leetcode"], B =
# ["ec","oc","ceo"]
# Output: ["facebook","leetcode"]
#
#
#
#
# Note:
#
#
#       1 <= A.length, B.length <= 10000
#       1 <= A[i].length, B[i].length <= 10
#       A[i] and B[i] consist only of lowercase letters.
#       All words in A[i] are unique: there isn't i != j with A[i] == A[j].# Time:  O(m + n)
# Space: O(1)

import collections


class Solution(object):
    def wordSubsets(self, A, B):
        """
        :type A: List[str]
        :type B: List[str]
        :rtype: List[str]
        """
        count = collections.Counter()
        for b in B:
```

```
    for c, n in collections.Counter(b).items():
        count[c] = max(count[c], n)
result = []
for a in A:
    count = collections.Counter(a)
    if all(count[c] >= count[c] for c in count):
        result.append(a)
return result
```

satisfiability-of-equality-equations.py

```python
# Example 2:
#
# Input: ["b==a","a==b"]
# Output: true
# Explanation: We could assign a = 1 and b = 1 to satisfy both equations.
#
#
#
# Example 3:
#
# Input: ["a==b","b==c","a==c"]
# Output: true
#
#
#
# Example 4:
#
# Input: ["a==b","b!=c","c==a"]
# Output: false
#
#
#
# Example 5:
#
# Input: ["c==c","b==d","x!=z"]
# Output: true
#
#
#
#
# Note:
#
#
#       1 <= equations.length <= 500
#       equations[i].length == 4
#       equations[i][0] and equations[i][3] are lowercase letters
#       equations[i][1] is either '=' or '!'
#       equations[i][2] is '='# Time:  O(n)
# Space: O(1)

class UnionFind(object):
    def __init__(self, n):
        self.set = range(n)

    def find_set(self, x):
        if self.set[x] != x:
            self.set[x] = self.find_set(self.set[x])  # path compression.
        return self.set[x]

    def union_set(self, x, y):
        x_root, y_root = map(self.find_set, (x, y))
        if x_root == y_root:
            return False
        self.set[min(x_root, y_root)] = max(x_root, y_root)
        return True


class Solution(object):
```

```python
    def equationsPossible(self, equations):
        """
        :type equations: List[str]
        :rtype: bool
        """
        union_find = UnionFind(26)
        for eqn in equations:
            x = ord(eqn[0]) - ord('a')
            y = ord(eqn[3]) - ord('a')
            if eqn[1] == '=':
                union_find.union_set(x, y)
        for eqn in equations:
            x = ord(eqn[0]) - ord('a')
            y = ord(eqn[3]) - ord('a')
            if eqn[1] == '!':
                if union_find.find_set(x) == union_find.find_set(y):
                    return False
        return True


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def equationsPossible(self, equations):
        """
        :type equations: List[str]
        :rtype: bool
        """
        graph = [[] for _ in xrange(26)]

        for eqn in equations:
            x = ord(eqn[0]) - ord('a')
            y = ord(eqn[3]) - ord('a')
            if eqn[1] == '!':
                if x == y:
                    return False
            else:
                graph[x].append(y)
                graph[y].append(x)

        color = [None]*26
        c = 0
        for i in xrange(26):
            if color[i] is not None:
                continue
            c += 1
            stack = [i]
            while stack:
                node = stack.pop()
                for nei in graph[node]:
                    if color[nei] is not None:
                        continue
                    color[nei] = c
                    stack.append(nei)

        for eqn in equations:
            if eqn[1] != '!':
                continue
            x = ord(eqn[0]) - ord('a')
            y = ord(eqn[3]) - ord('a')
```

```
    if color[x] is not None and \
       color[x] == color[y]:
        return False
return True
```

unique-substrings-in-wraparound-string.py

```
# Consider the string s to be the infinite wraparound string of
# "abcdefghijklmnopqrstuvwxyz", so s will look like this:
# "...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd....".
#
# Now we have another string p. Your job is to find out how many unique non-
# empty substrings of p are present in s. In particular, your input is the string
# p and you need to output the number of different non-empty substrings of p in
# the string s.
#
# Note: p consists of only lowercase English letters and the size of p might be
# over 10000.
#
# Example 1:
#
# Input: "a"
# Output: 1
#
# Explanation: Only the substring "a" of string "a" is in the string s.
#
#
#
# Example 2:
#
# Input: "cac"
# Output: 2
# Explanation: There are two substrings "a", "c" of string "cac" in the string
# s.
#
#
#
# Example 3:
#
# Input: "zab"
# Output: 6
# Explanation: There are six substrings "z", "a", "b", "za", "ab", "zab" of
# string "zab" in the string s.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def findSubstringInWraproundString(self, p):
        """
        :type p: str
        :rtype: int
        """
        letters = [0] * 26
        result, length = 0, 0
        for i in xrange(len(p)):
            curr = ord(p[i]) - ord('a')
            if i > 0 and ord(p[i-1]) != (curr-1)%26 + ord('a'):
                length = 0
            length += 1
            if length > letters[curr]:
                result += length - letters[curr]
                letters[curr] = length
        return result
```

exclusive-time-of-functions.py

```python
# On a single threaded CPU, we execute some functions.  Each function has a
# unique id between 0 and N-1.
#
# We store logs in timestamp order that describe when a function is entered or
# exited.
#
# Each log is a string with this format: "{function_id}:{"start" |
# "end"}:{timestamp}".  For example, "0:start:3" means the function with id 0
# started at the beginning of timestamp 3.  "1:end:2" means the function with id 1
# ended at the end of timestamp 2.
#
# A function's exclusive time is the number of units of time spent in this
# function.  Note that this does not include any recursive calls to child
# functions.
#
# The CPU is single threaded which means that only one function is being
# executed at a given time unit.
#
# Return the exclusive time of each function, sorted by their function id.
#
#
#
# Example 1:
#
#
#
# Input:
# n = 2
# logs = ["0:start:0","1:start:2","1:end:5","0:end:6"]
# Output: [3, 4]
# Explanation:
# Function 0 starts at the beginning of time 0, then it executes 2 units of time
# and reaches the end of time 1.
# Now function 1 starts at the beginning of time 2, executes 4 units of time and
# ends at time 5.
# Function 0 is running again at the beginning of time 6, and also ends at the
# end of time 6, thus executing for 1 unit of time.
# So function 0 spends 2 + 1 = 3 units of total time executing, and function 1
# spends 4 units of total time executing.
#
#
#
#
# Note:
#
#
#       1 <= n <= 100
#       Two functions won't start or end at the same time.
#       Functions will always log when they exit.# Time:  O(n)
# Space: O(n)

class Solution(object):
    def exclusiveTime(self, n, logs):
        """
        :type n: int
        :type logs: List[str]
        :rtype: List[int]
        """
```

```python
result = [0] * n
stk, prev = [], 0
for log in logs:
    tokens = log.split(":")
    if tokens[1] == "start":
        if stk:
            result[stk[-1]] += int(tokens[2]) - prev
        stk.append(int(tokens[0]))
        prev = int(tokens[2])
    else:
        result[stk.pop()] += int(tokens[2]) - prev + 1
        prev = int(tokens[2]) + 1
return result
```

flip-equivalent-binary-trees.py

```python
# For a binary tree T, we can define a flip operation as follows: choose any
# node, and swap the left and right child subtrees.
#
# A binary tree X is flip equivalent to a binary tree Y if and only if we can
# make X equal to Y after some number of flip operations.
#
# Given the roots of two binary trees root1 and root2, return true if the two
# trees are flip equivelent or false otherwise.
#
#
# Example 1:
#
# Input: root1 = [1,2,3,4,5,6,null,null,null,7,8], root2 =
# [1,3,2,null,6,4,5,null,null,null,null,8,7]
# Output: true
# Explanation: We flipped at nodes with values 1, 3, and 5.
#
#
# Example 2:
#
# Input: root1 = [], root2 = []
# Output: true
#
#
# Example 3:
#
# Input: root1 = [], root2 = [1]
# Output: false
#
#
# Example 4:
#
# Input: root1 = [0,null,1], root2 = []
# Output: false
#
#
# Example 5:
#
# Input: root1 = [0,null,1], root2 = [0,1]
# Output: true
#
#
#
# Constraints:
#
#
#       The number of nodes in each tree is in the range [0, 100].
#       Each value in each tree will be a unique integer in the range [0, 99].# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution(object):
```

```python
def flipEquiv(self, root1, root2):
    """
    :type root1: TreeNode
    :type root2: TreeNode
    :rtype: bool
    """
    if not root1 and not root2:
        return True
    if not root1 or not root2 or root1.val != root2.val:
        return False

    return (self.flipEquiv(root1.left, root2.left) and
            self.flipEquiv(root1.right, root2.right) or
            self.flipEquiv(root1.left, root2.right) and
            self.flipEquiv(root1.right, root2.left))
```

linked-list-cycle-ii.py

```
# Given a linked list, return the node where the cycle begins. If there is no
# cycle, return null.
#
# To represent a cycle in the given linked list, we use an integer pos which
# represents the position (0-indexed) in the linked list where tail connects to.
# If pos is -1, then there is no cycle in the linked list.
#
# Note: Do not modify the linked list.
#
#
#
# Example 1:
#
# Input: head = [3,2,0,-4], pos = 1
# Output: tail connects to node index 1
# Explanation: There is a cycle in the linked list, where tail connects to the
# second node.
#
#
#
#
# Example 2:
#
# Input: head = [1,2], pos = 0
# Output: tail connects to node index 0
# Explanation: There is a cycle in the linked list, where tail connects to the
# first node.
#
#
#
#
# Example 3:
#
# Input: head = [1], pos = -1
# Output: no cycle
# Explanation: There is no cycle in the linked list.
#
#
#
#
#
#
# Follow-up:
#
# Can you solve it without using extra space?# Time:  O(n)
# Space: O(1)

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

    def __str__(self):
        if self:
            return "{}".format(self.val)
        else:
            return None
```

```python
class Solution(object):
    # @param head, a ListNode
    # @return a list node
    def detectCycle(self, head):
        fast, slow = head, head
        while fast and fast.next:
            fast, slow = fast.next.next, slow.next
            if fast is slow:
                fast = head
                while fast is not slow:
                    fast, slow = fast.next, slow.next
                return fast
        return None
```

maximum-subarray-sum-with-one-deletion.py

```python
# Given an array of integers, return the maximum sum for a non-empty subarray
# (contiguous elements) with at most one element deletion. In other words, you
# want to choose a subarray and optionally delete one element from it so that
# there is still at least one element left and the sum of the remaining elements
# is maximum possible.
#
# Note that the subarray needs to be non-empty after deleting one element.
#
#
# Example 1:
#
# Input: arr = [1,-2,0,3]
# Output: 4
# Explanation: Because we can choose [1, -2, 0, 3] and drop -2, thus the
# subarray [1, 0, 3] becomes the maximum value.
#
# Example 2:
#
# Input: arr = [1,-2,-2,3]
# Output: 3
# Explanation: We just choose [3] and it's the maximum sum.
#
#
# Example 3:
#
# Input: arr = [-1,-1,-1,-1]
# Output: -1
# Explanation: The final subarray needs to be non-empty. You can't choose [-1]
# and delete -1 from it, then get an empty subarray to make the sum equals to 0.
#
#
#
# Constraints:
#
#
#       1 <= arr.length <= 10^5
#       -10^4 <= arr[i] <= 10^4class Solution(object):
    def maximumSum(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        result, prev, curr = float("-inf"), float("-inf"), float("-inf")
        for x in arr:
            curr = max(prev, curr+x, x)
            result = max(result, curr)
            prev = max(prev+x, x)
        return result
```

sort-colors.py

```python
# Given an array with n objects colored red, white or blue, sort them in-
# place 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.
#
# Example:
#
# Input: [2,0,2,1,1,0]
# Output: [0,0,1,1,2,2]
#
# 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 a one-pass algorithm using only constant space?# Time:  O(n)
# Space: O(1)

class Solution(object):
    def sortColors(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        def triPartition(nums, target):
            i, left, right = 0, 0, len(nums)-1
            while i <= right:
                if nums[i] > target:
                    nums[i], nums[right] = nums[right], nums[i]
                    right -= 1
                else:
                    if nums[i] < target:
                        nums[left], nums[i] = nums[i], nums[left]
                        left += 1
                    i += 1

        triPartition(nums, 1)
```

peeking-iterator.py

```python
# 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().
#
# 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.
#
#
# Follow up: How would you extend your design to be generic and work with all
# types, not just integer?# Time:  O(1) per peek(), next(), hasNext()
# Space: O(1)

class PeekingIterator(object):
    def __init__(self, iterator):
        """
        Initialize your data structure here.
        :type iterator: Iterator
        """
        self.iterator = iterator
        self.val_ = None
        self.has_next_ = iterator.hasNext()
        self.has_peeked_ = False


    def peek(self):
        """
        Returns the next element in the iteration without advancing the iterator.
        :rtype: int
        """
        if not self.has_peeked_:
            self.has_peeked_ = True
            self.val_ = self.iterator.next()
        return self.val_

    def next(self):
        """
        :rtype: int
        """
        self.val_ = self.peek()
        self.has_peeked_ = False
        self.has_next_ = self.iterator.hasNext()
        return self.val_

    def hasNext(self):
        """
        :rtype: bool
        """
        return self.has_next_
```

number-of-matching-subsequences.py

```python
# Given string S and a dictionary of words words, find the number of words[i]
# that is a subsequence of S.
#
# Example :
# Input:
# S = "abcde"
# words = ["a", "bb", "acd", "ace"]
# Output: 3
# Explanation: There are three words in words that are a subsequence of S: "a",
# "acd", "ace".
#
#
# Note:
#
#
#       All words in words and S will only consists of lowercase letters.
#       The length of S will be in the range of [1, 50000].
#       The length of words will be in the range of [1, 5000].
#       The length of words[i] will be in the range of [1, 50].# Time:  O(n + w), n is the size of S, w is the
# Space: O(k), k is the number of words

import collections


class Solution(object):
    def numMatchingSubseq(self, S, words):
        """
        :type S: str
        :type words: List[str]
        :rtype: int
        """
        waiting = collections.defaultdict(list)
        for word in words:
            it = iter(word)
            waiting[next(it, None)].append(it)
        for c in S:
            for it in waiting.pop(c, ()):
                waiting[next(it, None)].append(it)
        return len(waiting[None])
```

find-largest-value-in-each-tree-row.py

```python
# Given the root of a binary tree, return an array of the largest value in each
# row of the tree (0-indexed).
#
#
#
#
# Example 1:
#
# Input: root = [1,3,2,5,3,null,9]
# Output: [1,3,9]
#
#
# Example 2:
#
# Input: root = [1,2,3]
# Output: [1,3]
#
#
# Example 3:
#
# Input: root = [1]
# Output: [1]
#
#
# Example 4:
#
# Input: root = [1,null,2]
# Output: [1,2]
#
#
# Example 5:
#
# Input: root = []
# Output: []
#
#
#
# Constraints:
#
#
#       The number of the nodes in the tree will be in the range [1, 104].
#       -231 <= Node.val <= 231 - 1# Time:  O(n)
# Space: O(h)

class Solution(object):
    def largestValues(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        def largestValuesHelper(root, depth, result):
            if not root:
                return
            if depth == len(result):
                result.append(root.val)
            else:
                result[depth] = max(result[depth], root.val)
            largestValuesHelper(root.left, depth+1, result)
```

```python
            largestValuesHelper(root.right, depth+1, result)

        result = []
        largestValuesHelper(root, 0, result)
        return result


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def largestValues(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        result = []
        curr = [root]
        while any(curr):
            result.append(max(node.val for node in curr))
            curr = [child for node in curr for child in (node.left, node.right) if child]
        return result
```

contiguous-array.py

```python
# Given a binary array, find the maximum length of a contiguous subarray with
# equal number of 0 and 1.
#
#
# Example 1:
#
# Input: [0,1]
# Output: 2
# Explanation: [0, 1] is the longest contiguous subarray with equal number of 0
# and 1.
#
#
#
# Example 2:
#
# Input: [0,1,0]
# Output: 2
# Explanation: [0, 1] (or [1, 0]) is a longest contiguous subarray with equal
# number of 0 and 1.
#
#
#
# Note:
# The length of the given binary array will not exceed 50,000.# Time:  O(n)
# Space: O(n)

class Solution(object):
    def findMaxLength(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result, count = 0, 0
        lookup = {0: -1}
        for i, num in enumerate(nums):
            count += 1 if num == 1 else -1
            if count in lookup:
                result = max(result, i - lookup[count])
            else:
                lookup[count] = i

        return result
```

pacific-atlantic-water-flow.py

```python
# Given an m x n matrix of non-negative integers representing the height of each
# unit cell in a continent, the "Pacific ocean" touches the left and top edges of
# the matrix and the "Atlantic ocean" touches the right and bottom edges.
#
# Water can only flow in four directions (up, down, left, or right) from a cell
# to another one with height equal or lower.
#
# Find the list of grid coordinates where water can flow to both the Pacific and
# Atlantic ocean.
#
# Note:
#
#
#       The order of returned grid coordinates does not matter.
#       Both m and n are less than 150.
#
#
#
#
# Example:
#
# Given the following 5x5 matrix:
#
#   Pacific ~   ~   ~   ~   ~
#        ~  1   2   2   3  (5) *
#        ~  3   2   3  (4) (4) *
#        ~  2   4  (5)  3   1  *
#        ~ (6) (7)  1   4   5  *
#        ~ (5)  1   1   2   4  *
#           *   *   *   *   * Atlantic
#
# Return:
#
# [[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]] (positions with
# parentheses in above matrix).# Time:  O(m * n)
# Space: O(m * n)

class Solution(object):
    def pacificAtlantic(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: List[List[int]]
        """
        PACIFIC, ATLANTIC = 1, 2

        def pacificAtlanticHelper(matrix, x, y, prev_height, prev_val, visited, res):
            if (not 0 <= x < len(matrix)) or \
               (not 0 <= y < len(matrix[0])) or \
               matrix[x][y] < prev_height or \
               (visited[x][y] | prev_val) == visited[x][y]:
                return

            visited[x][y] |= prev_val
            if visited[x][y] == (PACIFIC | ATLANTIC):
                res.append((x, y))

            for d in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
                pacificAtlanticHelper(matrix, x + d[0], y + d[1], matrix[x][y], visited[x][y], visited, res)
```

```python
if not matrix:
    return []

res = []
m, n = len(matrix),len(matrix[0])
visited = [[0 for _ in xrange(n)] for _ in xrange(m)]

for i in xrange(m):
    pacificAtlanticHelper(matrix, i, 0, float("-inf"), PACIFIC, visited, res)
    pacificAtlanticHelper(matrix, i, n - 1, float("-inf"), ATLANTIC, visited, res)
for j in xrange(n):
    pacificAtlanticHelper(matrix, 0, j, float("-inf"), PACIFIC, visited, res)
    pacificAtlanticHelper(matrix, m - 1, j, float("-inf"), ATLANTIC, visited, res)

return res
```

minesweeper.py

```
# Let's play the minesweeper game (Wikipedia, online game)!
#
# You are given a 2D char matrix representing the game board. 'M' represents an
# unrevealed mine, 'E' represents an unrevealed empty square, 'B' represents a
# revealed blank square that has no adjacent (above, below, left, right, and all 4
# diagonals) mines, digit ('1' to '8') represents how many mines are adjacent to
# this revealed square, and finally 'X' represents a revealed mine.
#
# Now given the next click position (row and column indices) among all the
# unrevealed squares ('M' or 'E'), return the board after revealing this position
# according to the following rules:
#
#
#       If a mine ('M') is revealed, then the game is over - change it to 'X'.
#       If an empty square ('E') with no adjacent mines is revealed, then change
# it to revealed blank ('B') and all of its adjacent unrevealed squares should be
# revealed recursively.
#       If an empty square ('E') with at least one adjacent mine is revealed,
# then change it to a digit ('1' to '8') representing the number of adjacent
# mines.
#       Return the board when no more squares will be revealed.
#
#
#
#
# Example 1:
#
# Input:
#
# [['E', 'E', 'E', 'E', 'E'],
#  ['E', 'E', 'M', 'E', 'E'],
#  ['E', 'E', 'E', 'E', 'E'],
#  ['E', 'E', 'E', 'E', 'E']]
#
# Click : [3,0]
#
# Output:
#
# [['B', '1', 'E', '1', 'B'],
#  ['B', '1', 'M', '1', 'B'],
#  ['B', '1', '1', '1', 'B'],
#  ['B', 'B', 'B', 'B', 'B']]
#
# Explanation:
#
#
#
# Example 2:
#
# Input:
#
# [['B', '1', 'E', '1', 'B'],
#  ['B', '1', 'M', '1', 'B'],
#  ['B', '1', '1', '1', 'B'],
#  ['B', 'B', 'B', 'B', 'B']]
#
# Click : [1,2]
#
```

```python
# Output:
#
# [['B', '1', 'E', '1', 'B'],
#  ['B', '1', 'X', '1', 'B'],
#  ['B', '1', '1', '1', 'B'],
#  ['B', 'B', 'B', 'B', 'B']]
#
# Explanation:
#
#
#
#
#
# Note:
#
#
#       The range of the input matrix's height and width is [1,50].
#       The click position will only be an unrevealed square ('M' or 'E'), which
# also means the input board contains at least one clickable square.
#       The input board won't be a stage when game is over (some mines have been
# revealed).
#       For simplicity, not mentioned rules should be ignored in this problem.
# For example, you don't need to reveal all the unrevealed mines when the game is
# over, consider any cases that you will win the game or flag any squares.# Time:  O(m * n)
# Space: O(m + n)

import collections


class Solution(object):
    def updateBoard(self, board, click):
        """
        :type board: List[List[str]]
        :type click: List[int]
        :rtype: List[List[str]]
        """
        q = collections.deque([click])
        while q:
            row, col = q.popleft()
            if board[row][col] == 'M':
                board[row][col] = 'X'
            else:
                count = 0
                for i in xrange(-1, 2):
                    for j in xrange(-1, 2):
                        if i == 0 and j == 0:
                            continue
                        r, c = row + i, col + j
                        if not (0 <= r < len(board)) or not (0 <= c < len(board[r])):
                            continue
                        if board[r][c] == 'M' or board[r][c] == 'X':
                            count += 1

                if count:
                    board[row][col] = chr(count + ord('0'))
                else:
                    board[row][col] = 'B'
                    for i in xrange(-1, 2):
                        for j in xrange(-1, 2):
                            if i == 0 and j == 0:
```

```python
                            continue
                        r, c = row + i, col + j
                        if not (0 <= r < len(board)) or not (0 <= c < len(board[r])):
                            continue
                        if board[r][c] == 'E':
                            q.append((r, c))
                            board[r][c] = ' '

        return board


# Time:  O(m * n)
# Space: O(m * n)
class Solution2(object):
    def updateBoard(self, board, click):
        """
        :type board: List[List[str]]
        :type click: List[int]
        :rtype: List[List[str]]
        """
        row, col = click[0], click[1]
        if board[row][col] == 'M':
            board[row][col] = 'X'
        else:
            count = 0
            for i in xrange(-1, 2):
                for j in xrange(-1, 2):
                    if i == 0 and j == 0:
                        continue
                    r, c = row + i, col + j
                    if not (0 <= r < len(board)) or not (0 <= c < len(board[r])):
                        continue
                    if board[r][c] == 'M' or board[r][c] == 'X':
                        count += 1

            if count:
                board[row][col] = chr(count + ord('0'))
            else:
                board[row][col] = 'B'
                for i in xrange(-1, 2):
                    for j in xrange(-1, 2):
                        if i == 0 and j == 0:
                            continue
                        r, c = row + i, col + j
                        if not (0 <= r < len(board)) or not (0 <= c < len(board[r])):
                            continue
                        if board[r][c] == 'E':
                            self.updateBoard(board, (r, c))

        return board
```

count-numbers-with-unique-digits.py

```python
# Given a non-negative integer n, count all numbers with unique digits, x, where
# 0  x < 10n.
#
#
# Example:
#
# Input: 2
# Output: 91
# Explanation: The answer should be the total numbers in the range of 0  x <
# 100,
#              excluding 11,22,33,44,55,66,77,88,99
#
#
#
# Constraints:
#
#
#       0 <= n <= 8# Time:  O(n)
# Space: O(1)

class Solution(object):
    def countNumbersWithUniqueDigits(self, n):
        """
        :type n: int
        :rtype: int
        """
        if n == 0:
            return 1
        count, fk = 10, 9
        for k in xrange(2, n+1):
            fk *= 10 - (k-1)
            count += fk
        return count
```

most-stones-removed-with-same-row-or-column.py

```python
# Example 2:
#
# Input: stones = [[0,0],[0,2],[1,1],[2,0],[2,2]]
# Output: 3
#
#
#
# Example 3:
#
# Input: stones = [[0,0]]
# Output: 0
#
#
#
#
# Note:
#
#
#       1 <= stones.length <= 1000
#       0 <= stones[i][j] < 10000# Time:  O(n)
# Space: O(n)

class UnionFind(object):
    def __init__(self, n):
        self.set = range(n)

    def find_set(self, x):
        if self.set[x] != x:
            self.set[x] = self.find_set(self.set[x])  # path compression.
        return self.set[x]

    def union_set(self, x, y):
        x_root, y_root = map(self.find_set, (x, y))
        if x_root == y_root:
            return False
        self.set[min(x_root, y_root)] = max(x_root, y_root)
        return True


class Solution(object):
    def removeStones(self, stones):
        """
        :type stones: List[List[int]]
        :rtype: int
        """
        MAX_ROW = 10000
        union_find = UnionFind(2*MAX_ROW)
        for r, c in stones:
            union_find.union_set(r, c+MAX_ROW)
        return len(stones) - len({union_find.find_set(r) for r, _ in stones})
```

previous-permutation-with-one-swap.py

```
# Given an array A of positive integers (not necessarily distinct), return the
# lexicographically largest permutation that is smaller than A, that can be made
# with one swap (A swap exchanges the positions of two numbers A[i] and A[j]).  If
# it cannot be done, then return the same array.
#
#
#
# Example 1:
#
# Input: [3,2,1]
# Output: [3,1,2]
# Explanation: Swapping 2 and 1.
#
#
# Example 2:
#
# Input: [1,1,5]
# Output: [1,1,5]
# Explanation: This is already the smallest permutation.
#
#
# Example 3:
#
# Input: [1,9,4,6,7]
# Output: [1,7,4,6,9]
# Explanation: Swapping 9 and 7.
#
#
# Example 4:
#
# Input: [3,1,1,3]
# Output: [1,3,1,3]
# Explanation: Swapping 1 and 3.
#
#
#
#
# Note:
#
#
#       1 <= A.length <= 10000
#       1 <= A[i] <= 10000# Time:  O(n)
# Space: O(1)

class Solution(object):
    def prevPermOpt1(self, A):
        """
        :type A: List[int]
        :rtype: List[int]
        """
        for left in reversed(xrange(len(A)-1)):
            if A[left] > A[left+1]:
                break
        else:
            return A
        right = len(A)-1
        while A[left] <= A[right]:
            right -= 1
```

```
while A[right-1] == A[right]:
    right -= 1
A[left], A[right] = A[right], A[left]
return A
```

spiral-matrix.py

```python
# Given a matrix of m x n elements (m rows, n columns), return all elements of
# the matrix in spiral order.
#
# Example 1:
#
# Input:
# [
#  [ 1, 2, 3 ],
#  [ 4, 5, 6 ],
#  [ 7, 8, 9 ]
# ]
# Output: [1,2,3,6,9,8,7,4,5]
#
#
# Example 2:
# Input:
# [
#   [1, 2, 3, 4],
#   [5, 6, 7, 8],
#   [9,10,11,12]
# ]
# Output: [1,2,3,4,8,12,11,10,9,5,6,7]# Time:  O(m * n)
# Space: O(1)

class Solution(object):
    # @param matrix, a list of lists of integers
    # @return a list of integers
    def spiralOrder(self, matrix):
        result = []
        if matrix == []:
            return result

        left, right, top, bottom = 0, len(matrix[0]) - 1, 0, len(matrix) - 1

        while left <= right and top <= bottom:
            for j in xrange(left, right + 1):
                result.append(matrix[top][j])
            for i in xrange(top + 1, bottom):
                result.append(matrix[i][right])
            for j in reversed(xrange(left, right + 1)):
                if top < bottom:
                    result.append(matrix[bottom][j])
            for i in reversed(xrange(top + 1, bottom)):
                if left < right:
                    result.append(matrix[i][left])
            left, right, top, bottom = left + 1, right - 1, top + 1, bottom - 1

        return result
```

invalid-transactions.py

```python
# A transaction is possibly invalid if:
#
#
#       the amount exceeds $1000, or;
#       if it occurs within (and including) 60 minutes of another transaction
# with the same name in a different city.
#
#
# Each transaction string transactions[i] consists of comma separated values
# representing the name, time (in minutes), amount, and city of the transaction.
#
# Given a list of transactions, return a list of transactions that are possibly
# invalid.  You may return the answer in any order.
#
#
# Example 1:
#
# Input: transactions = ["alice,20,800,mtv","alice,50,100,beijing"]
# Output: ["alice,20,800,mtv","alice,50,100,beijing"]
# Explanation: The first transaction is invalid because the second transaction
# occurs within a difference of 60 minutes, have the same name and is in a
# different city. Similarly the second one is invalid too.
#
# Example 2:
#
# Input: transactions = ["alice,20,800,mtv","alice,50,1200,mtv"]
# Output: ["alice,50,1200,mtv"]
#
#
# Example 3:
#
# Input: transactions = ["alice,20,800,mtv","bob,50,1200,mtv"]
# Output: ["bob,50,1200,mtv"]
#
#
#
# Constraints:
#
#
#       transactions.length <= 1000
#       Each transactions[i] takes the form "{name},{time},{amount},{city}"
#       Each {name} and {city} consist of lowercase English letters, and have
# lengths between 1 and 10.
#       Each {time} consist of digits, and represent an integer between 0 and
# 1000.
#       Each {amount} consist of digits, and represent an integer between 0 and
# 2000.# Time:  O(nlogn)
# Space: O(n)


import collections


class Solution:
    def invalidTransactions(self, transactions):
        AMOUNT, MINUTES = 1000, 60
        trans = map(lambda x: (x[0], int(x[1]), int(x[2]), x[3]),
                    (transaction.split(',') for transaction in transactions))
```

```python
trans.sort(key=lambda t: t[1])
trans_indexes = collections.defaultdict(list)
for i, t in enumerate(trans):
    trans_indexes[t[0]].append(i)
result = []
for name, indexes in trans_indexes.iteritems():
    left, right = 0, 0
    for i, t_index in enumerate(indexes):
        t = trans[t_index]
        if (t[2] > AMOUNT):
            result.append("{},{},{},{}".format(*t))
            continue
        while left+1 < len(indexes) and trans[indexes[left]][1] < t[1]-MINUTES:
            left += 1
        while right+1 < len(indexes) and trans[indexes[right+1]][1] <= t[1]+MINUTES:
            right += 1
        for i in xrange(left, right+1):
            if trans[indexes[i]][3] != t[3]:
                result.append("{},{},{},{}".format(*t))
                break
return result
```

counting-bits.py

```python
# 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 1:
#
# Input: 2
# Output: [0,1,1]
#
# Example 2:
#
# Input: 5
# Output: [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.# Time:  O(n)
# Space: O(n)

class Solution(object):
    def countBits(self, num):
        """
        :type num: int
        :rtype: List[int]
        """
        res = [0]
        for i in xrange(1, num + 1):
            # Number of 1's in i = (i & 1) + number of 1's in (i / 2).
            res.append((i & 1) + res[i >> 1])
        return res

    def countBits2(self, num):
        """
        :type num: int
        :rtype: List[int]
        """
        s = [0]
        while len(s) <= num:
            s.extend(map(lambda x: x + 1, s))
        return s[:num + 1]
```

stone-game-ii.py

```python
# Alex and Lee continue their games with piles of stones.  There are a number
# of piles arranged in a row, and each pile has a positive integer number of
# stones piles[i].  The objective of the game is to end with the most stones.
#
# Alex and Lee take turns, with Alex starting first.  Initially, M = 1.
#
# On each player's turn, that player can take all the stones in the first X
# remaining piles, where 1 <= X <= 2M.  Then, we set M = max(M, X).
#
# The game continues until all the stones have been taken.
#
# Assuming Alex and Lee play optimally, return the maximum number of stones Alex
# can get.
#
#
# Example 1:
#
# Input: piles = [2,7,9,4,4]
# Output: 10
# Explanation:  If Alex takes one pile at the beginning, Lee takes two piles,
# then Alex takes 2 piles again. Alex can get 2 + 4 + 4 = 10 piles in total. If
# Alex takes two piles at the beginning, then Lee can take all three piles left.
# In this case, Alex get 2 + 7 = 9 piles in total. So we return 10 since it's
# larger.
#
#
#
# Constraints:
#
#
#       1 <= piles.length <= 100
#       1 <= piles[i] <= 10 ^ 4# Time:  O(n*(logn)^2)
# Space: O(nlogn)

class Solution(object):
    def stoneGameII(self, piles):
        """
        :type piles: List[int]
        :rtype: int
        """
        def dp(piles, lookup, i, m):
            if i+2*m >= len(piles):
                return piles[i]
            if (i, m) not in lookup:
                lookup[i, m] = piles[i] - \
                               min(dp(piles, lookup, i+x, max(m, x))
                                   for x in xrange(1, 2*m+1))
            return lookup[i, m]

        for i in reversed(xrange(len(piles)-1)):
            piles[i] += piles[i+1]
        return dp(piles, {}, 0, 1)
```

lru-cache.py

```python
# Design and implement a data structure for Least Recently Used (LRU) cache. It
# should support the following operations: get and put.
#
# get(key) - Get the value (will always be positive) of the key if the key
# exists in the cache, otherwise return -1.
#
# put(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.
#
# The cache is initialized with a positive capacity.
#
# Follow up:
#
# Could you do both operations in O(1) time complexity?
#
# Example:
#
# LRUCache cache = new LRUCache( 2 /* capacity */ );
#
# cache.put(1, 1);
# cache.put(2, 2);
# cache.get(1);       // returns 1
# cache.put(3, 3);    // evicts key 2
# cache.get(2);       // returns -1 (not found)
# cache.put(4, 4);    // evicts key 1
# cache.get(1);       // returns -1 (not found)
# cache.get(3);       // returns 3
# cache.get(4);       // returns 4# Time:  O(1), per operation.
# Space: O(k), k is the capacity of cache.

class ListNode(object):
    def __init__(self, key, val):
        self.val = val
        self.key = key
        self.next = None
        self.prev = None

class LinkedList(object):
    def __init__(self):
        self.head = None
        self.tail = None

    def insert(self, node):
        node.next, node.prev = None, None  # avoid dirty node
        if self.head is None:
            self.head = node
        else:
            self.tail.next = node
            node.prev = self.tail
        self.tail = node

    def delete(self, node):
        if node.prev:
            node.prev.next = node.next
        else:
            self.head = node.next
        if node.next:
```

```python
            node.next.prev = node.prev
        else:
            self.tail = node.prev
        node.next, node.prev = None, None  # make node clean

class LRUCache(object):

    # @param capacity, an integer
    def __init__(self, capacity):
        self.list = LinkedList()
        self.dict = {}
        self.capacity = capacity

    def _insert(self, key, val):
        node = ListNode(key, val)
        self.list.insert(node)
        self.dict[key] = node


    # @return an integer
    def get(self, key):
        if key in self.dict:
            val = self.dict[key].val
            self.list.delete(self.dict[key])
            self._insert(key, val)
            return val
        return -1


    # @param key, an integer
    # @param value, an integer
    # @return nothing
    def put(self, key, val):
        if key in self.dict:
            self.list.delete(self.dict[key])
        elif len(self.dict) == self.capacity:
            del self.dict[self.list.head.key]
            self.list.delete(self.list.head)
        self._insert(key, val)


import collections
class LRUCache2(object):
    def __init__(self, capacity):
        self.cache = collections.OrderedDict()
        self.capacity = capacity

    def get(self, key):
        if key not in self.cache:
            return -1
        val = self.cache[key]
        del self.cache[key]
        self.cache[key] = val
        return val

    def put(self, key, value):
        if key in self.cache:
            del self.cache[key]
        elif len(self.cache) == self.capacity:
            self.cache.popitem(last=False)
```

```
self.cache[key] = value
```

maximum-product-of-splitted-binary-tree.py

```python
# Given a binary tree root. Split the binary tree into two subtrees by
# removing 1 edge such that the product of the sums of the subtrees are maximized.
#
# Since the answer may be too large, return it modulo 10^9 + 7.
#
#
# Example 1:
#
#
#
# Input: root = [1,2,3,4,5,6]
# Output: 110
# Explanation: Remove the red edge and get 2 binary trees with sum 11 and 10.
# Their product is 110 (11*10)
#
#
# Example 2:
#
#
#
# Input: root = [1,null,2,3,4,null,null,5,6]
# Output: 90
# Explanation:  Remove the red edge and get 2 binary trees with sum 15 and
# 6.Their product is 90 (15*6)
#
#
# Example 3:
#
# Input: root = [2,3,9,10,7,8,6,5,4,11,1]
# Output: 1025
#
#
# Example 4:
#
# Input: root = [1,1]
# Output: 1
#
#
#
# Constraints:
#
#
#       Each tree has at most 50000 nodes and at least 2 nodes.
#       Each node's value is between [1, 10000].# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def maxProduct(self, root):
        """
        :type root: TreeNode
```

```python
    :rtype: int
    """
    MOD = 10**9 + 7
    def dfs(root, total, result):
        if not root:
            return 0
        subtotal = dfs(root.left, total, result)+dfs(root.right, total, result)+root.val
        result[0] = max(result[0], subtotal*(total-subtotal) )
        return subtotal

    result = [0]
    dfs(root, dfs(root, 0, result), result)
    return result[0] % MOD
```

valid-tic-tac-toe-state.py

```python
# A Tic-Tac-Toe board is given as a string array board. Return True if and only
# if it is possible to reach this board position during the course of a valid tic-
# tac-toe game.
#
# The board is a 3 x 3 array, and consists of characters " ", "X", and "O".  The
# " " character represents an empty square.
#
# Here are the rules of Tic-Tac-Toe:
#
#
#       Players take turns placing characters into empty squares (" ").
#       The first player always places "X" characters, while the second player
# always places "O" characters.
#       "X" and "O" characters are always placed into empty squares, never
# filled ones.
#       The game ends when there are 3 of the same (non-empty) character filling
# any row, column, or diagonal.
#       The game also ends if all squares are non-empty.
#       No more moves can be played if the game is over.
#
#
# Example 1:
# Input: board = ["O  ", "   ", "   "]
# Output: false
# Explanation: The first player always plays "X".
#
# Example 2:
# Input: board = ["XOX", " X ", "   "]
# Output: false
# Explanation: Players take turns making moves.
#
# Example 3:
# Input: board = ["XXX", "   ", "OOO"]
# Output: false
#
# Example 4:
# Input: board = ["XOX", "O O", "XOX"]
# Output: true
#
#
# Note:
#
#
#       board is a length-3 array of strings, where each string board[i] has
# length 3.
#       Each board[i][j] is a character in the set {" ", "X", "O"}.# Time:  O(1)
# Space: O(1)

class Solution(object):
    def validTicTacToe(self, board):
        """
        :type board: List[str]
        :rtype: bool
        """
        def win(board, player):
            for i in xrange(3):
                if all(board[i][j] == player for j in xrange(3)):
                    return True
```

```
        if all(board[j][i] == player for j in xrange(3)):
            return True

    return (player == board[1][1] == board[0][0] == board[2][2] or \
            player == board[1][1] == board[0][2] == board[2][0])

FIRST, SECOND = ('X', 'O')
x_count = sum(row.count(FIRST) for row in board)
o_count = sum(row.count(SECOND) for row in board)
if o_count not in {x_count-1, x_count}: return False
if win(board, FIRST) and x_count-1 != o_count: return False
if win(board, SECOND) and x_count != o_count: return False

return True
```

decrease-elements-to-make-array-zigzag.py

```python
# Given an array nums of integers, a move consists of choosing any element and
# decreasing it by 1.
#
# An array A is a zigzag array if either:
#
#
#       Every even-indexed element is greater than adjacent elements, ie. A[0] >
# A[1] < A[2] > A[3] < A[4] > ...
#       OR, every odd-indexed element is greater than adjacent elements,
# ie. A[0] < A[1] > A[2] < A[3] > A[4] < ...
#
#
# Return the minimum number of moves to transform the given array nums into a
# zigzag array.
#
#
# Example 1:
#
# Input: nums = [1,2,3]
# Output: 2
# Explanation: We can decrease 2 to 0 or 3 to 1.
#
#
# Example 2:
#
# Input: nums = [9,6,1,6,2]
# Output: 4
#
#
#
# Constraints:
#
#
#       1 <= nums.length <= 1000
#       1 <= nums[i] <= 1000# Time:  O(n)
# Space: O(1)

class Solution(object):
    def movesToMakeZigzag(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result = [0, 0]
        for i in xrange(len(nums)):
            left = nums[i-1] if i-1 >= 0 else float("inf")
            right = nums[i+1] if i+1 < len(nums) else float("inf")
            result[i%2] += max(nums[i] - min(left, right) + 1, 0)
        return min(result)
```

maximum-xor-of-two-numbers-in-an-array.py

```python
# Given a non-empty array of numbers, a0, a1, a2, ... , an-1, where 0  ai < 231.
#
# Find the maximum result of ai XOR aj, where 0  i, j < n.
#
# Could you do this in O(n) runtime?
#
# Example:
#
# Input: [3, 10, 5, 25, 2, 8]
#
# Output: 28
#
# Explanation: The maximum result is 5 ^ 25 = 28.# Time:  O(n)
# Space: O(n)

class Solution(object):
    def findMaximumXOR(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result = 0

        for i in reversed(xrange(32)):
            result <<= 1
            prefixes = set()
            for n in nums:
                prefixes.add(n >> i)
            for p in prefixes:
                if (result | 1) ^ p in prefixes:
                    result += 1
                    break

        return result
```

longest-common-subsequence.py

```
# Given two strings text1 and text2, return the length of their longest common
# subsequence.
#
# A subsequence of a string is a new string generated from the original string
# with some characters(can be none) deleted without changing the relative order of
# the remaining characters. (eg, "ace" is a subsequence of "abcde" while "aec" is
# not). A common subsequence of two strings is a subsequence that is common to
# both strings.
#
#
#
# If there is no common subsequence, return 0.
#
#
# Example 1:
#
# Input: text1 = "abcde", text2 = "ace"
# Output: 3
# Explanation: The longest common subsequence is "ace" and its length is 3.
#
#
# Example 2:
#
# Input: text1 = "abc", text2 = "abc"
# Output: 3
# Explanation: The longest common subsequence is "abc" and its length is 3.
#
#
# Example 3:
#
# Input: text1 = "abc", text2 = "def"
# Output: 0
# Explanation: There is no such common subsequence, so the result is 0.
#
#
#
# Constraints:
#
#
#       1 <= text1.length <= 1000
#       1 <= text2.length <= 1000
#       The input strings consist of lowercase English characters only.# Time:  O(m * n)
# Space: O(min(m, n))

class Solution(object):
    def longestCommonSubsequence(self, text1, text2):
        """
        :type text1: str
        :type text2: str
        :rtype: int
        """
        if len(text1) < len(text2):
            return self.longestCommonSubsequence(text2, text1)

        dp = [[0 for _ in xrange(len(text2)+1)] for _ in xrange(2)]
        for i in xrange(1, len(text1)+1):
            for j in xrange(1, len(text2)+1):
                dp[i%2][j] = dp[(i-1)%2][j-1]+1 if text1[i-1] == text2[j-1] \
```

```
                        else max(dp[(i-1)%2][j], dp[i%2][j-1])
return dp[len(text1)%2][len(text2)]
```

binary-tree-inorder-traversal.py

```python
# Given a binary tree, return the inorder traversal of its nodes' values.
#
# Example:
#
# Input: [1,null,2,3]
#    1
#     \
#      2
#     /
#    3
#
# Output: [1,3,2]
#
# Follow up: Recursive solution is trivial, could you do it iteratively?# Time:  O(n)
# Space: O(1)

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


# Morris Traversal Solution
class Solution(object):
    def inorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        result, curr = [], root
        while curr:
            if curr.left is None:
                result.append(curr.val)
                curr = curr.right
            else:
                node = curr.left
                while node.right and node.right != curr:
                    node = node.right

                if node.right is None:
                    node.right = curr
                    curr = curr.left
                else:
                    result.append(curr.val)
                    node.right = None
                    curr = curr.right

        return result


# Time:  O(n)
# Space: O(h)
# Stack Solution
class Solution2(object):
    def inorderTraversal(self, root):
        """
        :type root: TreeNode
```

```
    :rtype: List[int]
    """
    result, stack = [], [(root, False)]
    while stack:
        root, is_visited = stack.pop()
        if root is None:
            continue
        if is_visited:
            result.append(root.val)
        else:
            stack.append((root.right, False))
            stack.append((root, True))
            stack.append((root.left, False))
    return result
```

sort-the-matrix-diagonally.py

```python
# Given a m * n matrix mat of integers, sort it diagonally in ascending order
# from the top-left to the bottom-right then return the sorted array.
#
#
# Example 1:
#
# Input: mat = [[3,3,1,1],[2,2,1,2],[1,1,1,2]]
# Output: [[1,1,1,1],[1,2,2,2],[1,2,3,3]]
#
#
#
# Constraints:
#
#
#       m == mat.length
#       n == mat[i].length
#       1 <= m, n <= 100
#       1 <= mat[i][j] <= 100# Time:  O(m * n * log(min(m, n))
# Space: O(m * n)

import collections


class Solution(object):
    def diagonalSort(self, mat):
        """
        :type mat: List[List[int]]
        :rtype: List[List[int]]
        """
        lookup = collections.defaultdict(list)
        for i in xrange(len(mat)):
            for j in xrange(len(mat[0])):
                lookup[i-j].append(mat[i][j])
        for v in lookup.itervalues():
            v.sort()
        for i in reversed(xrange(len(mat))):
            for j in reversed(xrange(len(mat[0]))):
                mat[i][j] = lookup[i-j].pop()
        return mat
```

flip-binary-tree-to-match-preorder-traversal.py

```
# Example 2:
#
#
#
# Input: root = [1,2,3], voyage = [1,3,2]
# Output: [1]
#
#
#
# Example 3:
#
#
#
# Input: root = [1,2,3], voyage = [1,2,3]
# Output: []
#
#
#
#
# Note:
#
#
#       1 <= N <= 100# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def flipMatchVoyage(self, root, voyage):
        """
        :type root: TreeNode
        :type voyage: List[int]
        :rtype: List[int]
        """
        def dfs(root, voyage, i, result):
            if not root:
                return True
            if root.val != voyage[i[0]]:
                return False
            i[0] += 1
            if root.left and root.left.val != voyage[i[0]]:
                result.append(root.val)
                return dfs(root.right, voyage, i, result) and \
                       dfs(root.left, voyage, i, result)
            return dfs(root.left, voyage, i, result) and \
                   dfs(root.right, voyage, i, result)

        result = []
        return result if dfs(root, voyage, [0], result) else [-1]
```

bulb-switcher-ii.py

```python
# There is a room with n lights which are turned on initially and 4 buttons on
# the wall. After performing exactly m unknown operations towards buttons, you
# need to return how many different kinds of status of the n lights could be.
#
# Suppose n lights are labeled as number [1, 2, 3 ..., n], function of these 4
# buttons are given below:
#
#
#       Flip all the lights.
#       Flip lights with even numbers.
#       Flip lights with odd numbers.
#       Flip lights with (3k + 1) numbers, k = 0, 1, 2, ...
#
#
#
#
# Example 1:
#
# Input: n = 1, m = 1.
# Output: 2
# Explanation: Status can be: [on], [off]
#
#
#
#
# Example 2:
#
# Input: n = 2, m = 1.
# Output: 3
# Explanation: Status can be: [on, off], [off, on], [off, off]
#
#
#
#
# Example 3:
#
# Input: n = 3, m = 1.
# Output: 4
# Explanation: Status can be: [off, on, off], [on, off, on], [off, off, off],
# [off, on, on].
#
#
#
#
# Note: n and m both fit in range [0, 1000].# Time:  O(1)
# Space: O(1)

class Solution(object):
    def flipLights(self, n, m):
        """
        :type n: int
        :type m: int
        :rtype: int
        """
        if m == 0:
            return 1
        if n == 1:
            return 2
```

```
if m == 1 and n == 2:
    return 3
if m == 1 or n == 2:
    return 4
if m == 2:
    return 7
return 8
```

arithmetic-slices.py

```python
# A sequence of numbers is called arithmetic if it consists of at least three
# elements and if the difference between any two consecutive elements is the same.
#
# For example, these are arithmetic sequences:
#
# 1, 3, 5, 7, 9
# 7, 7, 7, 7
# 3, -1, -5, -9
#
# The following sequence is not arithmetic.
#
# 1, 1, 2, 5, 7
#
#
# A zero-indexed array A consisting of N numbers is given. A slice of that array
# is any pair of integers (P, Q) such that 0 <= P < Q < N.
#
# A slice (P, Q) of the array A is called arithmetic if the sequence:
#
# A[P], A[P + 1], ..., A[Q - 1], A[Q] is arithmetic. In particular, this means
# that P + 1 < Q.
#
# The function should return the number of arithmetic slices in the array A.
#
#
# Example:
#
# A = [1, 2, 3, 4]
#
# return: 3, for 3 arithmetic slices in A: [1, 2, 3], [2, 3, 4] and [1, 2, 3, 4]
# itself.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numberOfArithmeticSlices(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        res, i = 0, 0
        while i+2 < len(A):
            start = i
            while i+2 < len(A) and A[i+2] + A[i] == 2*A[i+1]:
                res += i - start + 1
                i += 1
            i += 1

        return res
```

unique-paths.py

```python
# 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 7 x 3 grid. How many possible unique paths are there?
#
#
# Example 1:
#
# Input: m = 3, n = 2
# Output: 3
# Explanation:
# From the top-left corner, there are a total of 3 ways to reach the bottom-
# right corner:
# 1. Right -> Right -> Down
# 2. Right -> Down -> Right
# 3. Down -> Right -> Right
#
#
# Example 2:
#
# Input: m = 7, n = 3
# Output: 28
#
#
#
# Constraints:
#
#
#       1 <= m, n <= 100
#       It's guaranteed that the answer will be less than or equal to 2 * 10 ^
# 9.# Time:  O(m * n)
# Space: O(m + n)

class Solution(object):
    # @return an integer
    def uniquePaths(self, m, n):
        if m < n:
            return self.uniquePaths(n, m)
        ways = [1] * n

        for i in xrange(1, m):
            for j in xrange(1, n):
                ways[j] += ways[j - 1]

        return ways[n - 1]
```

bag-of-tokens.py

```python
# Example 2:
#
# Input: tokens = [100,200], P = 150
# Output: 1
#
#
#
# Example 3:
#
# Input: tokens = [100,200,300,400], P = 200
# Output: 2
#
#
#
#
# Note:
#
#
#       tokens.length <= 1000
#       0 <= tokens[i] < 10000
#       0 <= P < 10000# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def bagOfTokensScore(self, tokens, P):
        """
        :type tokens: List[int]
        :type P: int
        :rtype: int
        """
        tokens.sort()
        result, points = 0, 0
        left, right = 0, len(tokens)-1
        while left <= right:
            if P >= tokens[left]:
                P -= tokens[left]
                left += 1
                points += 1
                result = max(result, points)
            elif points > 0:
                points -= 1
                P += tokens[right]
                right -= 1
            else:
                break
        return result
```

decode-ways.py

```python
# A message containing letters from A-Z is being encoded to numbers using the
# following mapping:
#
# 'A' -> 1
# 'B' -> 2
# ...
# 'Z' -> 26
#
#
# Given a non-empty string containing only digits, determine the total number of
# ways to decode it.
#
# Example 1:
#
# Input: "12"
# Output: 2
# Explanation: It could be decoded as "AB" (1 2) or "L" (12).
#
#
# Example 2:
#
# Input: "226"
# Output: 3
# Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2
# 6).# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numDecodings(self, s):
        """
        :type s: str
        :rtype: int
        """
        if len(s) == 0 or s[0] == '0':
            return 0
        prev, prev_prev = 1, 0
        for i in xrange(len(s)):
            cur = 0
            if s[i] != '0':
                cur = prev
            if i > 0 and (s[i - 1] == '1' or (s[i - 1] == '2' and s[i] <= '6')):
                cur += prev_prev
            prev, prev_prev = cur, prev
        return prev
```

statistics-from-a-large-sample.py

```python
# We sampled integers between 0 and 255, and stored the results in an array
# count:  count[k] is the number of integers we sampled equal to k.
#
# Return the minimum, maximum, mean, median, and mode of the sample
# respectively, as an array of floating point numbers.  The mode is guaranteed to
# be unique.
#
# (Recall that the median of a sample is:
#
#
#       The middle element, if the elements of the sample were sorted and the
# number of elements is odd;
#       The average of the middle two elements, if the elements of the sample
# were sorted and the number of elements is even.)
#
#
#
# Example 1:
# Input: count = [0,1,3,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
# 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
# 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
# 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
# 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
# 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
# 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
# Output: [1.00000,3.00000,2.37500,2.50000,3.00000]
# Example 2:
# Input: count = [0,4,3,2,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
# 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
# 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
# 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
# 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
# 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
# 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
# Output: [1.00000,4.00000,2.18182,2.00000,1.00000]
#
#
# Constraints:
#
#
#       count.length == 256
#       1 <= sum(count) <= 10^9
#       The mode of the sample that count represents is unique.
#       Answers within 10^-5 of the true value will be accepted as correct.# Time:  O(n)
# Space: O(1)

import bisect


class Solution(object):
    def sampleStats(self, count):
        """
        :type count: List[int]
        :rtype: List[float]
        """
        n = sum(count)
        mi = next(i for i in xrange(len(count)) if count[i]) * 1.0
        ma = next(i for i in reversed(xrange(len(count))) if count[i]) * 1.0
```

```
mean = sum(i * v for i, v in enumerate(count)) * 1.0 / n
mode = count.index(max(count)) * 1.0
for i in xrange(1, len(count)):
    count[i] += count[i-1]
median1 = bisect.bisect_left(count, (n+1) // 2)
median2 = bisect.bisect_left(count, (n+2) // 2)
median = (median1+median2) / 2.0
return [mi, ma, mean, median, mode]
```

array-nesting.py

```python
# A zero-indexed array A of length N contains all integers from 0 to N-1. Find
# and return the longest length of set S, where S[i] = {A[i], A[A[i]], A[A[A[i]]],
# ... } subjected to the rule below.
#
# Suppose the first element in S starts with the selection of element A[i] of
# index = i, the next element in S should be A[A[i]], and then A[A[A[i]]]... By that
# analogy, we stop adding right before a duplicate element occurs in S.
#
#
#
# Example 1:
#
# Input: A = [5,4,0,3,1,6,2]
# Output: 4
# Explanation:
# A[0] = 5, A[1] = 4, A[2] = 0, A[3] = 3, A[4] = 1, A[5] = 6, A[6] = 2.
#
# One of the longest S[K]:
# S[0] = {A[0], A[5], A[6], A[2]} = {5, 6, 2, 0}
#
#
#
#
# Note:
#
#
#       N is an integer within the range [1, 20,000].
#       The elements of A are all distinct.
#       Each element of A is an integer within the range [0, N-1].# Time:  O(n)
# Space: O(1)

class Solution(object):
    def arrayNesting(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result = 0
        for num in nums:
            if num is not None:
                start, count = num, 0
                while nums[start] is not None:
                    temp = start
                    start = nums[start]
                    nums[temp] = None
                    count += 1
                result = max(result, count)
        return result
```

divide-two-integers.py

```python
# Given two integers dividend and divisor, divide two integers without using
# multiplication, division and mod operator.
#
# Return the quotient after dividing dividend by divisor.
#
# The integer division should truncate toward zero, which means losing its
# fractional part. For example, truncate(8.345) = 8 and truncate(-2.7335) = -2.
#
# Example 1:
#
# Input: dividend = 10, divisor = 3
# Output: 3
# Explanation: 10/3 = truncate(3.33333..) = 3.
#
#
# Example 2:
#
# Input: dividend = 7, divisor = -3
# Output: -2
# Explanation: 7/-3 = truncate(-2.33333..) = -2.
#
#
# Note:
#
#
#       Both dividend and divisor will be 32-bit signed integers.
#       The divisor will never be 0.
#       Assume we are dealing with an environment which could only store
# integers within the 32-bit signed integer range: [231,  231  1]. For the
# purpose of this problem, assume that your function returns 231  1 when the
# division result overflows.# Time:  O(logn) = O(1)
# Space: O(1)

class Solution(object):
    def divide(self, dividend, divisor):
        """
        :type dividend: int
        :type divisor: int
        :rtype: int
        """
        result, dvd, dvs = 0, abs(dividend), abs(divisor)
        while dvd >= dvs:
            inc = dvs
            i = 0
            while dvd >= inc:
                dvd -= inc
                result += 1 << i
                inc <<= 1
                i += 1
        if dividend > 0 and divisor < 0 or dividend < 0 and divisor > 0:
            return -result
        else:
            return result

    def divide2(self, dividend, divisor):
        """
        :type dividend: int
        :type divisor: int
```

```python
        :rtype: int
        """
        positive = (dividend < 0) is (divisor < 0)
        dividend, divisor = abs(dividend), abs(divisor)
        res = 0
        while dividend >= divisor:
            temp, i = divisor, 1
            while dividend >= temp:
                dividend -= temp
                res += i
                i <<= 1
                temp <<= 1
        if not positive:
            res = -res
        return min(max(-2147483648, res), 2147483647)
```

coin-change.py

```
# 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:
#
# Input: coins = [1, 2, 5], amount = 11
# Output: 3
# Explanation: 11 = 5 + 5 + 1
#
# Example 2:
#
# Input: coins = [2], amount = 3
# Output: -1
#
#
# Note:
#
# You may assume that you have an infinite number of each kind of coin.# Time:  O(n * k), n is the number of c
# Space: O(k)

class Solution(object):
    def coinChange(self, coins, amount):
        """
        :type coins: List[int]
        :type amount: int
        :rtype: int
        """
        INF = 0x7fffffff  # Using float("inf") would be slower.
        amounts = [INF] * (amount + 1)
        amounts[0] = 0
        for i in xrange(amount + 1):
            if amounts[i] != INF:
                for coin in coins:
                    if i + coin <= amount:
                        amounts[i + coin] = min(amounts[i + coin], amounts[i] + 1)
        return amounts[amount] if amounts[amount] != INF else -1
```

regions-cut-by-slashes.py

```python
# Example 3:
#
# Input:
# [
#   "\\/",
#   "/\\"
# ]
# Output: 4
# Explanation: (Recall that because \ characters are escaped, "\\/" refers to
# \/, and "/\\" refers to /\.)
# The 2x2 grid is as follows:
#
#
#
#
# Example 4:
#
# Input:
# [
#   "/\\",
#   "\\/"
# ]
# Output: 5
# Explanation: (Recall that because \ characters are escaped, "/\\" refers to
# /\, and "\\/" refers to \/.)
# The 2x2 grid is as follows:
#
#
#
#
# Example 5:
#
# Input:
# [
#   "//",
#   "/ "
# ]
# Output: 3
# Explanation: The 2x2 grid is as follows:
#
#
#
#
#
# Note:
#
#
#       1 <= grid.length == grid[0].length <= 30
#       grid[i][j] is either '/', '\', or ' '.# Time:  O(n^2)
# Space: O(n^2)

class UnionFind(object):
    def __init__(self, n):
        self.set = range(n)
        self.count = n

    def find_set(self, x):
       if self.set[x] != x:
```

```python
            self.set[x] = self.find_set(self.set[x])  # path compression.
        return self.set[x]

    def union_set(self, x, y):
        x_root, y_root = map(self.find_set, (x, y))
        if x_root != y_root:
            self.set[min(x_root, y_root)] = max(x_root, y_root)
            self.count -= 1


class Solution(object):
    def regionsBySlashes(self, grid):
        """
        :type grid: List[str]
        :rtype: int
        """
        def index(n, i, j, k):
            return (i*n + j)*4 + k

        union_find = UnionFind(len(grid)**2 * 4)
        N, E, S, W = range(4)
        for i in xrange(len(grid)):
            for j in xrange(len(grid)):
                if i:
                    union_find.union_set(index(len(grid), i-1, j, S),
                                         index(len(grid),i, j, N))
                if j:
                    union_find.union_set(index(len(grid), i, j-1, E),
                                         index(len(grid), i, j, W))
                if grid[i][j] != "/":
                    union_find.union_set(index(len(grid), i, j, N),
                                         index(len(grid), i, j, E))
                    union_find.union_set(index(len(grid), i, j, S),
                                         index(len(grid), i, j, W))
                if grid[i][j] != "\\":
                    union_find.union_set(index(len(grid), i, j, W),
                                         index(len(grid), i, j, N))
                    union_find.union_set(index(len(grid), i, j, E),
                                         index(len(grid), i, j, S))
        return union_find.count
```

letter-combinations-of-a-phone-number.py

```
# Given a string containing digits from 2-9 inclusive, 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. Note that 1 does not map to any letters.
#
#
#
# Example:
#
# Input: "23"
# 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.# Time:  O(n * 4^n)
# Space: O(n)

class Solution(object):
    # @return a list of strings, [s1, s2]
    def letterCombinations(self, digits):
        if not digits:
            return []

        lookup, result = ["", "", "abc", "def", "ghi", "jkl", "mno", \
                          "pqrs", "tuv", "wxyz"], [""]

        for digit in reversed(digits):
            choices = lookup[int(digit)]
            m, n = len(choices), len(result)
            result += [result[i % n] for i in xrange(n, m * n)]

            for i in xrange(m * n):
                result[i] = choices[i / n] + result[i]

        return result


# Time:  O(n * 4^n)
# Space: O(n)
# Recursive Solution
class Solution2(object):
    # @return a list of strings, [s1, s2]
    def letterCombinations(self, digits):
        if not digits:
            return []
        lookup, result = ["", "", "abc", "def", "ghi", "jkl", "mno", \
                          "pqrs", "tuv", "wxyz"], []
        self.letterCombinationsRecu(result, digits, lookup, "", 0)
        return result

    def letterCombinationsRecu(self, result, digits, lookup, cur, n):
        if n == len(digits):
            result.append(cur)
        else:
            for choice in lookup[int(digits[n])]:
```

```python
            self.letterCombinationsRecu(result, digits, lookup, cur + choice, n + 1)
```

longest-repeating-character-replacement.py

```python
# Given a string s that consists of only uppercase English letters, you can
# perform at most k operations on that string.
#
# In one operation, you can choose any character of the string and change it to
# any other uppercase English character.
#
# Find the length of the longest sub-string containing all repeating letters you
# can get after performing the above operations.
#
# Note:
#
# Both the string's length and k will not exceed 104.
#
# Example 1:
#
# Input:
# s = "ABAB", k = 2
#
# Output:
# 4
#
# Explanation:
# Replace the two 'A's with two 'B's or vice versa.
#
#
#
#
# Example 2:
#
# Input:
# s = "AABABBA", k = 1
#
# Output:
# 4
#
# Explanation:
# Replace the one 'A' in the middle with 'B' and form "AABBBBA".
# The substring "BBBB" has the longest repeating letters, which is 4.# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def characterReplacement(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: int
        """
        result, max_count = 0, 0
        count = collections.Counter()
        for i in xrange(len(s)):
            count[s[i]] += 1
            max_count = max(max_count, count[s[i]])
            if result - max_count >= k:
                count[s[i-result]] -= 1
            else:
```

```
        result += 1
return result
```

search-in-rotated-sorted-array.py

```python
# Given an integer array nums sorted in ascending order, and an integer target.
#
# Suppose that nums 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 should search for target in nums and if you found return its index,
# otherwise return -1.
#
#
# Example 1:
# Input: nums = [4,5,6,7,0,1,2], target = 0
# Output: 4
# Example 2:
# Input: nums = [4,5,6,7,0,1,2], target = 3
# Output: -1
# Example 3:
# Input: nums = [1], target = 0
# Output: -1
#
#
# Constraints:
#
#
#       1 <= nums.length <= 5000
#       -10^4 <= nums[i] <= 10^4
#       All values of nums are unique.
#       nums is guranteed to be rotated at some pivot.
#       -10^4 <= target <= 10^4# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def search(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        left, right = 0, len(nums) - 1

        while left <= right:
            mid = left + (right - left) / 2

            if nums[mid] == target:
                return mid
            elif (nums[mid] >= nums[left] and nums[left] <= target < nums[mid]) or \
                 (nums[mid] < nums[left] and not (nums[mid] < target <= nums[right])):
                right = mid - 1
            else:
                left = mid + 1

        return -1
```

increasing-triplet-subsequence.py

```python
# Given an unsorted array return whether an increasing subsequence of length 3
# exists or not in the array.
#
# Formally the function should:
#
# Return true if there exists i, j, k
#
# such that arr[i] < arr[j] < arr[k] given 0  i < j < k  n-1 else return
# false.
#
# Note: Your algorithm should run in O(n) time complexity and O(1) space
# complexity.
#
#
# Example 1:
#
# Input: [1,2,3,4,5]
# Output: true
#
#
#
# Example 2:
#
# Input: [5,4,3,2,1]
# Output: false# Time:  O(n)
# Space: O(1)

import bisect


class Solution(object):
    def increasingTriplet(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        min_num, a, b = float("inf"), float("inf"), float("inf")
        for c in nums:
            if min_num >= c:
                min_num = c
            elif b >= c:
                a, b = min_num, c
            else:  # a < b < c
                return True
        return False

# Time:  O(n * logk)
# Space: O(k)
# Generalization of k-uplet.
class Solution_Generalization(object):
    def increasingTriplet(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        def increasingKUplet(nums, k):
            inc = [float('inf')] * (k - 1)
            for num in nums:
```

```
            i = bisect.bisect_left(inc, num)
            if i >= k - 1:
                return True
            inc[i] = num
        return k == 0

    return increasingKUplet(nums, 3)
```

permutations-ii.py

```python
# Given a collection of numbers that might contain duplicates, return all
# possible unique permutations.
#
# Example:
#
# Input: [1,1,2]
# Output:
# [
#   [1,1,2],
#   [1,2,1],
#   [2,1,1]
# ]# Time:  O(n * n!)
# Space: O(n)

class Solution(object):
    def permuteUnique(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        nums.sort()
        result = []
        used = [False] * len(nums)
        self.permuteUniqueRecu(result, used, [], nums)
        return result

    def permuteUniqueRecu(self, result, used, cur, nums):
        if len(cur) == len(nums):
            result.append(cur + [])
            return
        for i in xrange(len(nums)):
            if used[i] or (i > 0 and nums[i-1] == nums[i] and not used[i-1]):
                continue
            used[i] = True
            cur.append(nums[i])
            self.permuteUniqueRecu(result, used, cur, nums)
            cur.pop()
            used[i] = False

class Solution2(object):
    # @param num, a list of integer
    # @return a list of lists of integers
    def permuteUnique(self, nums):
        solutions = [[]]

        for num in nums:
            next = []
            for solution in solutions:
                for i in xrange(len(solution) + 1):
                    candidate = solution[:i] + [num] + solution[i:]
                    if candidate not in next:
                        next.append(candidate)

            solutions = next

        return solutions
```

find-and-replace-pattern.py

```python
# You have a list of words and a pattern, and you want to know which words in
# words matches the pattern.
#
# A word matches the pattern if there exists a permutation of letters p so that
# after replacing every letter x in the pattern with p(x), we get the desired
# word.
#
# (Recall that a permutation of letters is a bijection from letters to letters:
# every letter maps to another letter, and no two letters map to the same letter.)
#
# Return a list of the words in words that match the given pattern.
#
# You may return the answer in any order.
#
#
#
#
# Example 1:
#
# Input: words = ["abc","deq","mee","aqq","dkd","ccc"], pattern = "abb"
# Output: ["mee","aqq"]
# Explanation: "mee" matches the pattern because there is a permutation {a -> m,
# b -> e, ...}.
# "ccc" does not match the pattern because {a -> c, b -> c, ...} is not a
# permutation,
# since a and b map to the same letter.
#
#
#
# Note:
#
#
#       1 <= words.length <= 50
#       1 <= pattern.length = words[i].length <= 20# Time:  O(n * l)
# Space: O(1)

import itertools


class Solution(object):
    def findAndReplacePattern(self, words, pattern):
        """
        :type words: List[str]
        :type pattern: str
        :rtype: List[str]
        """
        def match(word):
            lookup = {}
            for x, y in itertools.izip(pattern, word):
                if lookup.setdefault(x, y) != y:
                    return False
            return len(set(lookup.values())) == len(lookup.values())

        return filter(match, words)
```

time-based-key-value-store.py

```
# Create a timebased key-value store class TimeMap, that supports two
# operations.
#
# 1. set(string key, string value, int timestamp)
#
#
#       Stores the key and value, along with the given timestamp.
#
#
# 2. get(string key, int timestamp)
#
#
#       Returns a value such that set(key, value, timestamp_prev) was called
# previously, with timestamp_prev <= timestamp.
#       If there are multiple such values, it returns the one with the largest
# timestamp_prev.
#       If there are no values, it returns the empty string ("").
#
#
#
#
#
# Example 1:
#
# Input: inputs = ["TimeMap","set","get","get","set","get","get"], inputs =
# [[],["foo","bar",1],["foo",1],["foo",3],["foo","bar2",4],["foo",4],["foo",5]]
# Output: [null,null,"bar","bar",null,"bar2","bar2"]
# Explanation:
# TimeMap kv;
# kv.set("foo", "bar", 1); // store the key "foo" and value "bar" along with
# timestamp = 1
# kv.get("foo", 1);  // output "bar"
# kv.get("foo", 3); // output "bar" since there is no value corresponding to foo
# at timestamp 3 and timestamp 2, then the only value is at timestamp 1 ie "bar"
# kv.set("foo", "bar2", 4);
# kv.get("foo", 4); // output "bar2"
# kv.get("foo", 5); //output "bar2"
#
#
#
#
# Example 2:
#
# Input: inputs = ["TimeMap","set","set","get","get","get","get","get"], inputs
# = [[],["love","high",10],["love","low",20],["love",5],["love",10],["love",15],["
# love",20],["love",25]]
# Output: [null,null,null,"","high","high","low","low"]
#
#
#
#
#
#
# Note:
#
#
#       All key/value strings are lowercase.
#       All key/value strings have length in the range [1, 100]
```

```python
#       The timestamps for all TimeMap.set operations are strictly increasing.
#       1 <= timestamp <= 10^7
#       TimeMap.set and TimeMap.get functions will be called a total of 120000
# times (combined) per test case.# Time:  set: O(1)
#        get: O(logn)
# Space: O(n)

import collections
import bisect


class TimeMap(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.lookup = collections.defaultdict(list)

    def set(self, key, value, timestamp):
        """
        :type key: str
        :type value: str
        :type timestamp: int
        :rtype: None
        """
        self.lookup[key].append((timestamp, value))


    def get(self, key, timestamp):
        """
        :type key: str
        :type timestamp: int
        :rtype: str
        """
        A = self.lookup.get(key, None)
        if A is None:
            return ""
        i = bisect.bisect_right(A, (timestamp+1, 0))
        return A[i-1][1] if i else ""


# Your TimeMap object will be instantiated and called as such:
# obj = TimeMap()
# obj.set(key,value,timestamp)
# param_2 = obj.get(key,timestamp)
```

ugly-number-iii.py

```python
# Write a program to find the n-th ugly number.
#
# Ugly numbers are positive integers which are divisible by a or b or c.
#
#
# Example 1:
#
# Input: n = 3, a = 2, b = 3, c = 5
# Output: 4
# Explanation: The ugly numbers are 2, 3, 4, 5, 6, 8, 9, 10... The 3rd is 4.
#
# Example 2:
#
# Input: n = 4, a = 2, b = 3, c = 4
# Output: 6
# Explanation: The ugly numbers are 2, 3, 4, 6, 8, 9, 10, 12... The 4th is 6.
#
#
# Example 3:
#
# Input: n = 5, a = 2, b = 11, c = 13
# Output: 10
# Explanation: The ugly numbers are 2, 4, 6, 8, 10, 11, 12, 13... The 5th is 10.
#
#
# Example 4:
#
# Input: n = 1000000000, a = 2, b = 217983653, c = 336916467
# Output: 1999999984
#
#
#
# Constraints:
#
#
#       1 <= n, a, b, c <= 10^9
#       1 <= a * b * c <= 10^18
#       It's guaranteed that the result will be in range [1, 2 * 10^9]# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def nthUglyNumber(self, n, a, b, c):
        """
        :type n: int
        :type a: int
        :type b: int
        :type c: int
        :rtype: int
        """
        def gcd(a, b):
            while b:
                a, b = b, a % b
            return a

        def lcm(x, y):
            return x//gcd(x, y)*y

        def count(x, a, b, c, lcm_a_b, lcm_b_c, lcm_c_a, lcm_a_b_c):
```

```python
    return x//a + x//b + x//c - (x//lcm_a_b + x//lcm_b_c + x//lcm_c_a) + x//lcm_a_b_c

lcm_a_b, lcm_b_c, lcm_c_a = lcm(a, b), lcm(b, c), lcm(c, a)
lcm_a_b_c = lcm(lcm_a_b, lcm_b_c)

left, right = 1, 2*10**9
while left <= right:
    mid = left + (right-left)//2
    if count(mid, a, b, c, lcm_a_b, lcm_b_c, lcm_c_a, lcm_a_b_c) >= n:
        right = mid-1
    else:
        left = mid+1
return left
```

utf-8-validation.py

```python
# A character in UTF8 can be from 1 to 4 bytes long, subjected to the following
# rules:
#
# For 1-byte character, the first bit is a 0, followed by its unicode code.
# For n-bytes character, the first n-bits are all one's, the n+1 bit is 0,
# followed by n-1 bytes with most significant 2 bits being 10.
#
# This is how the UTF-8 encoding would work:
#
#    Char. number range  |        UTF-8 octet sequence
#       (hexadecimal)    |              (binary)
#    --------------------+---------------------------------------------
#    0000 0000-0000 007F | 0xxxxxxx
#    0000 0080-0000 07FF | 110xxxxx 10xxxxxx
#    0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
#    0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
#
#
# Given an array of integers representing the data, return whether it is a valid
# utf-8 encoding.
#
#
# Note:
#
# The input is an array of integers. Only the least significant 8 bits of each
# integer is used to store the data. This means each integer represents only 1
# byte of data.
#
#
#
# Example 1:
# data = [197, 130, 1], which represents the octet sequence: 11000101 10000010
# 00000001.
#
# Return true.
# It is a valid utf-8 encoding for a 2-bytes character followed by a 1-byte
# character.
#
#
#
#
# Example 2:
# data = [235, 140, 4], which represented the octet sequence: 11101011 10001100
# 00000100.
#
# Return false.
# The first 3 bits are all one's and the 4th bit is 0 means it is a 3-bytes
# character.
# The next byte is a continuation byte which starts with 10 and that's correct.
# But the second continuation byte does not start with 10, so it is invalid.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def validUtf8(self, data):
        """
        :type data: List[int]
        :rtype: bool
        """
```

```python
count = 0
for c in data:
    if count == 0:
        if (c >> 5) == 0b110:
            count = 1
        elif (c >> 4) == 0b1110:
            count = 2
        elif (c >> 3) == 0b11110:
            count = 3
        elif (c >> 7):
            return False
    else:
        if (c >> 6) != 0b10:
            return False
        count -= 1
return count == 0
```

longest-increasing-subsequence.py

```
# Given an unsorted array of integers, find the length of longest increasing
# subsequence.
#
# Example:
#
# Input: [10,9,2,5,3,7,101,18]
# Output: 4
# Explanation: The longest increasing subsequence is [2,3,7,101], therefore the
# length is 4.
#
# Note:
#
#
#       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?# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def lengthOfLIS(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        LIS = []
        def insert(target):
            left, right = 0, len(LIS) - 1
            # Find the first index "left" which satisfies LIS[left] >= target
            while left <= right:
                mid = left + (right - left) // 2
                if LIS[mid] >= target:
                    right = mid - 1
                else:
                    left = mid + 1
            # If not found, append the target.
            if left == len(LIS):
                LIS.append(target)
            else:
                LIS[left] = target

        for num in nums:
            insert(num)

        return len(LIS)

# Time:  O(n^2)
# Space: O(n)
# Traditional DP solution.
class Solution2(object):
    def lengthOfLIS(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        dp = []  # dp[i]: the length of LIS ends with nums[i]
```

```python
for i in xrange(len(nums)):
    dp.append(1)
    for j in xrange(i):
        if nums[j] < nums[i]:
            dp[i] = max(dp[i], dp[j] + 1)
return max(dp) if dp else 0
```

last-stone-weight-ii.py

```python
# We have a collection of rocks, each rock has a positive integer weight.
#
# Each turn, we choose any two rocks and smash them together.  Suppose the
# stones have weights x and y with x <= y.  The result of this smash is:
#
#
#       If x == y, both stones are totally destroyed;
#       If x != y, the stone of weight x is totally destroyed, and the stone of
# weight y has new weight y-x.
#
#
# At the end, there is at most 1 stone left.  Return the smallest possible
# weight of this stone (the weight is 0 if there are no stones left.)
#
#
#
# Example 1:
#
# Input: [2,7,4,1,8,1]
# Output: 1
# Explanation:
# We can combine 2 and 4 to get 2 so the array converts to [2,7,1,8,1] then,
# we can combine 7 and 8 to get 1 so the array converts to [2,1,1,1] then,
# we can combine 2 and 1 to get 1 so the array converts to [1,1,1] then,
# we can combine 1 and 1 to get 0 so the array converts to [1] then that's the
# optimal value.
#
#
#
#
# Note:
#
#
#       1 <= stones.length <= 30
#       1 <= stones[i] <= 100# Time:  O(2^n)
# Space: O(2^n)

class Solution(object):
    def lastStoneWeightII(self, stones):
        """
        :type stones: List[int]
        :rtype: int
        """
        dp = {0}
        for stone in stones:
            dp |= {stone+i for i in dp}
        S = sum(stones)
        return min(abs(i-(S-i)) for i in dp)
```

largest-number.py

```python
# Given a list of non negative integers, arrange them such that they form the
# largest number.
#
# Example 1:
#
# Input: [10,2]
# Output: "210"
#
# Example 2:
#
# Input: [3,30,34,5,9]
# Output: "9534330"
#
#
# Note: The result may be very large, so you need to return a string instead of
# an integer.# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    # @param num, a list of integers
    # @return a string
    def largestNumber(self, num):
        num = [str(x) for x in num]
        num.sort(cmp=lambda x, y: cmp(y + x, x + y))
        largest = ''.join(num)
        return largest.lstrip('0') or '0'
```

coloring-a-border.py

```python
# Example 3:
#
# Input: grid = [[1,1,1],[1,1,1],[1,1,1]], r0 = 1, c0 = 1, color = 2
# Output: [[2, 2, 2], [2, 1, 2], [2, 2, 2]]# Time:  O(m * n)
# Space: O(m + n)

import collections


class Solution(object):
    def colorBorder(self, grid, r0, c0, color):
        """
        :type grid: List[List[int]]
        :type r0: int
        :type c0: int
        :type color: int
        :rtype: List[List[int]]
        """
        directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]

        lookup, q, borders = set([(r0, c0)]), collections.deque([(r0, c0)]), []
        while q:
            r, c = q.popleft()
            is_border = False

            for direction in directions:
                nr, nc = r+direction[0], c+direction[1]
                if not ((0 <= nr < len(grid)) and \
                        (0 <= nc < len(grid[0])) and \
                        grid[nr][nc] == grid[r][c]):
                    is_border = True
                    continue
                if (nr, nc) in lookup:
                    continue
                lookup.add((nr, nc))
                q.append((nr, nc))

            if is_border:
                borders.append((r, c))

        for r, c in borders:
            grid[r][c] = color
        return grid
```

validate-stack-sequences.py

```python
# Example 2:
#
# Input: pushed = [1,2,3,4,5], popped = [4,3,5,1,2]
# Output: false
# Explanation: 1 cannot be popped before 2.# Time:  O(n)
# Space: O(n)

class Solution(object):
    def validateStackSequences(self, pushed, popped):
        """
        :type pushed: List[int]
        :type popped: List[int]
        :rtype: bool
        """
        i = 0
        s = []
        for v in pushed:
            s.append(v)
            while s and i < len(popped) and s[-1] == popped[i]:
                s.pop()
                i += 1
        return i == len(popped)
```

4sum.py

```python
# Given an array nums of n integers and an integer target, are there elements a,
# b, c, and d in nums 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.
#
# Example:
#
# Given array nums = [1, 0, -1, 0, -2, 2], and target = 0.
#
# A solution set is:
# [
#   [-1,  0, 0, 1],
#   [-2, -1, 1, 2],
#   [-2,  0, 0, 2]
# ]# Time:  O(n^3)
# Space: O(1)

import collections


# Two pointer solution. (1356ms)
class Solution(object):
    def fourSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        nums.sort()
        res = []
        for i in xrange(len(nums) - 3):
            if i and nums[i] == nums[i - 1]:
                continue
            for j in xrange(i + 1, len(nums) - 2):
                if j != i + 1 and nums[j] == nums[j - 1]:
                    continue
                sum = target - nums[i] - nums[j]
                left, right = j + 1, len(nums) - 1
                while left < right:
                    if nums[left] + nums[right] == sum:
                        res.append([nums[i], nums[j], nums[left], nums[right]])
                        right -= 1
                        left += 1
                        while left < right and nums[left] == nums[left - 1]:
                            left += 1
                        while left < right and nums[right] == nums[right + 1]:
                            right -= 1
                    elif nums[left] + nums[right] > sum:
                        right -= 1
                    else:
                        left += 1
        return res


# Time:  O(n^2 * p)
```

```python
# Space: O(n^2 * p)
# Hash solution. (224ms)
class Solution2(object):
    def fourSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        nums, result, lookup = sorted(nums), [], collections.defaultdict(list)
        for i in xrange(0, len(nums) - 1):
            for j in xrange(i + 1, len(nums)):
                is_duplicated = False
                for [x, y] in lookup[nums[i] + nums[j]]:
                    if nums[x] == nums[i]:
                        is_duplicated = True
                        break
                if not is_duplicated:
                    lookup[nums[i] + nums[j]].append([i, j])
        ans = {}
        for c in xrange(2, len(nums)):
            for d in xrange(c+1, len(nums)):
                if target - nums[c] - nums[d] in lookup:
                    for [a, b] in lookup[target - nums[c] - nums[d]]:
                        if b < c:
                            quad = [nums[a], nums[b], nums[c], nums[d]]
                            quad_hash = " ".join(str(quad))
                            if quad_hash not in ans:
                                ans[quad_hash] = True
                                result.append(quad)
        return result


# Time:  O(n^2 * p) ~ O(n^4)
# Space: O(n^2)
class Solution3(object):
    def fourSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[List[int]]
        """
        nums, result, lookup = sorted(nums), [], collections.defaultdict(list)
        for i in xrange(0, len(nums) - 1):
            for j in xrange(i + 1, len(nums)):
                lookup[nums[i] + nums[j]].append([i, j])

        for i in lookup.keys():
            if target - i in lookup:
                for x in lookup[i]:
                    for y in lookup[target - i]:
                        [a, b], [c, d] = x, y
                        if a is not c and a is not d and \
                           b is not c and b is not d:
                            quad = sorted([nums[a], nums[b], nums[c], nums[d]])
                            if quad not in result:
                                result.append(quad)
        return sorted(result)
```

bulb-switcher.py

```python
# 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 i-th round, you
# toggle every i bulb. For the n-th round, you only toggle the last bulb. Find how
# many bulbs are on after n rounds.
#
# Example:
#
# Input: 3
# Output: 1
# Explanation:
# At first, the three bulbs are [off, off, off].
# After first round, the three bulbs are [on, on, on].
# After second round, the three bulbs are [on, off, on].
# After third round, the three bulbs are [on, off, off].
#
# So you should return 1, because there is only one bulb is on.# Time:  O(1)
# Space: O(1)

import math


class Solution(object):
    def bulbSwitch(self, n):
        """
        type n: int
        rtype: int
        """
        # The number of full squares.
        return int(math.sqrt(n))
```

path-in-zigzag-labelled-binary-tree.py

```python
# In an infinite binary tree where every node has two children, the nodes are
# labelled in row order.
#
# In the odd numbered rows (ie., the first, third, fifth,...), the labelling is
# left to right, while in the even numbered rows (second, fourth, sixth,...), the
# labelling is right to left.
#
#
#
# Given the label of a node in this tree, return the labels in the path from the
# root of the tree to the node with that label.
#
#
# Example 1:
#
# Input: label = 14
# Output: [1,3,4,14]
#
#
# Example 2:
#
# Input: label = 26
# Output: [1,2,6,10,26]
#
#
#
# Constraints:
#
#
#       1 <= label <= 10^6# Time:  O(logn)
# Space: O(logn)

class Solution(object):
    def pathInZigZagTree(self, label):
        """
        :type label: int
        :rtype: List[int]
        """
        count = 2**label.bit_length()
        result = []
        while label >= 1:
            result.append(label)
            label = ((count//2) + ((count-1)-label)) // 2
            count //= 2
        result.reverse()
        return result
```

break-a-palindrome.py

```python
# Given a palindromic string palindrome, replace exactly one character by any
# lowercase English letter so that the string becomes the lexicographically
# smallest possible string that isn't a palindrome.
#
# After doing so, return the final string.  If there is no way to do so, return
# the empty string.
#
#
# Example 1:
#
# Input: palindrome = "abccba"
# Output: "aaccba"
#
#
# Example 2:
#
# Input: palindrome = "a"
# Output: ""
#
#
#
# Constraints:
#
#
#       1 <= palindrome.length <= 1000
#       palindrome consists of only lowercase English letters.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def breakPalindrome(self, palindrome):
        """
        :type palindrome: str
        :rtype: str
        """
        for i in xrange(len(palindrome)//2):
            if palindrome[i] != 'a':
                return palindrome[:i] + 'a' + palindrome[i+1:]
        return palindrome[:-1] + 'b' if len(palindrome) >= 2 else ""
```

balance-a-binary-search-tree.py

```
# Given a binary search tree, return a balanced binary search tree with the same
# node values.
#
# A binary search tree is balanced if and only if the depth of the two subtrees
# of every node never differ by more than 1.
#
# If there is more than one answer, return any of them.
#
#
# Example 1:
#
#
#
# Input: root = [1,null,2,null,3,null,4,null,null]
# Output: [2,1,3,null,null,null,4]
# Explanation: This is not the only correct answer, [3,1,4,null,2,null,null] is
# also correct.
#
#
#
# Constraints:
#
#
#       The number of nodes in the tree is between 1 and 10^4.
#       The tree nodes will have distinct values between 1 and 10^5.# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


# dfs solution with stack
class Solution(object):
    def balanceBST(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        def inorderTraversal(root):
            result, stk = [], [(root, False)]
            while stk:
                node, is_visited = stk.pop()
                if node is None:
                    continue
                if is_visited:
                    result.append(node.val)
                else:
                    stk.append((node.right, False))
                    stk.append((node, True))
                    stk.append((node.left, False))
            return result

        def sortedArrayToBst(arr):
            ROOT, LEFT, RIGHT = range(3)
```

```python
            result = [None]
            stk = [(0, len(arr), ROOT, result)]
            while stk:
                i, j, update, ret = stk.pop()
                if i >= j:
                    continue
                mid = i + (j-i)//2
                node = TreeNode(arr[mid])
                if update == ROOT:
                    ret[0] = node
                elif update == LEFT:
                    ret[0].left = node
                else:
                    ret[0].right = node
                stk.append((mid+1, j, RIGHT, [node]))
                stk.append((i, mid, LEFT, [node]))
            return result[0]

        return sortedArrayToBst(inorderTraversal(root))


# Time:  O(n)
# Space: O(h)
# dfs solution with recursion
class Solution2(object):
    def balanceBST(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
        """
        def inorderTraversalHelper(node, arr):
            if not node:
                return
            inorderTraversalHelper(node.left, arr)
            arr.append(node.val)
            inorderTraversalHelper(node.right, arr)

        def sortedArrayToBstHelper(arr, i, j):
            if i >= j:
                return None
            mid = i + (j-i)//2
            node = TreeNode(arr[mid])
            node.left = sortedArrayToBstHelper(arr, i, mid)
            node.right = sortedArrayToBstHelper(arr, mid+1, j)
            return node

        arr = []
        inorderTraversalHelper(root, arr)
        return sortedArrayToBstHelper(arr, 0, len(arr))
```

verify-preorder-serialization-of-a-binary-tree.py

```
# One way to serialize a binary tree is to use pre-order traversal. When we
# encounter a non-null node, we record the node's value. If it is a null node, we
# record using a sentinel value such as #.
#
#      _9_
#     /   \
#    3     2
#   / \   / \
#  4   1  #  6
# / \ / \   / \
# # # # #   # #
#
#
# For example, the above binary tree can be serialized to the string
# "9,3,4,#,#,1,#,#,2,#,6,#,#", where # represents a null node.
#
# Given a string of comma separated values, verify whether it is a correct
# preorder traversal serialization of a binary tree. Find an algorithm without
# reconstructing the tree.
#
# 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:
#
# Input: "9,3,4,#,#,1,#,#,2,#,6,#,#"
# Output: true
#
# Example 2:
#
# Input: "1,#"
# Output: false
#
#
# Example 3:
#
# Input: "9,#,#,1"
# Output: false# Time:  O(n)
# Space: O(1)

class Solution(object):
    def isValidSerialization(self, preorder):
        """
        :type preorder: str
        :rtype: bool
        """
        def split_iter(s, tok):
            start = 0
            for i in xrange(len(s)):
                if s[i] == tok:
                    yield s[start:i]
                    start = i + 1
            yield s[start:]

        if not preorder:
```

```python
    return False

depth, cnt = 0, preorder.count(',') + 1
for tok in split_iter(preorder, ','):
    cnt -= 1
    if tok == "#":
        depth -= 1
        if depth < 0:
            break
    else:
        depth += 1
return cnt == 0 and depth < 0
```

valid-triangle-number.py

```python
# Given an array consists of non-negative integers,  your task is to count the
# number of triplets chosen from the array that can make triangles if we take them
# as side lengths of a triangle.
#
# Example 1:
#
# Input: [2,2,3,4]
# Output: 3
# Explanation:
# Valid combinations are:
# 2,3,4 (using the first 2)
# 2,3,4 (using the second 2)
# 2,2,3
#
#
#
# Note:
#
#
# The length of the given array won't exceed 1000.
# The integers in the given array are in the range of [0, 1000].# Time:  O(n^2)
# Space: O(1)

class Solution(object):
    def triangleNumber(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result = 0
        nums.sort()
        for i in xrange(len(nums)-2):
            if nums[i] == 0:
                continue
            k = i+2
            for j in xrange(i+1, len(nums)-1):
                while k < len(nums) and nums[i] + nums[j] > nums[k]:
                    k += 1
                result += k-j-1
        return result
```

coin-change-2.py

```python
# You are given coins of different denominations and a total amount of money.
# Write a function to compute the number of combinations that make up that amount.
# You may assume that you have infinite number of each kind of coin.
#
#
#
#
#
#
# Example 1:
#
# Input: amount = 5, coins = [1, 2, 5]
# Output: 4
# Explanation: there are four ways to make up the amount:
# 5=5
# 5=2+2+1
# 5=2+1+1+1
# 5=1+1+1+1+1
#
#
# Example 2:
#
# Input: amount = 3, coins = [2]
# Output: 0
# Explanation: the amount of 3 cannot be made up just with coins of 2.
#
#
# Example 3:
#
# Input: amount = 10, coins = [10]
# Output: 1
#
#
#
#
# Note:
#
# You can assume that
#
#
#       0 <= amount <= 5000
#       1 <= coin <= 5000
#       the number of coins is less than 500
#       the answer is guaranteed to fit into signed 32-bit integer# Time:  O(n * m)
# Space: O(m)

class Solution(object):
    def change(self, amount, coins):
        """
        :type amount: int
        :type coins: List[int]
        :rtype: int
        """
        dp = [0] * (amount+1)
        dp[0] = 1
        for coin in coins:
            for i in xrange(coin, amount+1):
                dp[i] += dp[i-coin]
```

```
    return dp[amount]
```

subsets.py

```python
# Given a set of distinct integers, nums, return all possible subsets (the power
# set).
#
# Note: The solution set must not contain duplicate subsets.
#
# Example:
#
# Input: nums = [1,2,3]
# Output:
# [
#   [3],
#   [1],
#   [2],
#   [1,2,3],
#   [1,3],
#   [2,3],
#   [1,2],
#   []
# ]# Time:  O(n * 2^n)
# Space: O(1)

class Solution(object):
    def subsets(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        nums.sort()
        result = [[]]
        for i in xrange(len(nums)):
            size = len(result)
            for j in xrange(size):
                result.append(list(result[j]))
                result[-1].append(nums[i])
        return result


# Time:  O(n * 2^n)
# Space: O(1)
class Solution2(object):
    def subsets(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        result = []
        i, count = 0, 1 << len(nums)
        nums.sort()

        while i < count:
            cur = []
            for j in xrange(len(nums)):
                if i & 1 << j:
                    cur.append(nums[j])
            result.append(cur)
            i += 1

        return result
```

```python
# Time:  O(n * 2^n)
# Space: O(1)
class Solution3(object):
    def subsets(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        return self.subsetsRecu([], sorted(nums))

    def subsetsRecu(self, cur, nums):
        if not nums:
            return [cur]

        return self.subsetsRecu(cur, nums[1:]) + self.subsetsRecu(cur + [nums[0]], nums[1:])
```

kth-smallest-element-in-a-sorted-matrix.py

```python
# Given a n x n matrix where each of the rows and columns are sorted in
# ascending order, find the kth smallest element in the matrix.
#
#
# Note that it is the kth smallest element in the sorted order, not the kth
# distinct element.
#
#
# Example:
# matrix = [
#    [ 1,  5,  9],
#    [10, 11, 13],
#    [12, 13, 15]
# ],
# k = 8,
#
# return 13.
#
#
#
# Note:
#
# You may assume k is always valid, 1  k  n2.# Time:  O(k * log(min(n, m, k))), with n x m matrix
# Space: O(min(n, m, k))

from heapq import heappush, heappop

class Solution(object):
    def kthSmallest(self, matrix, k):
        """
        :type matrix: List[List[int]]
        :type k: int
        :rtype: int
        """
        kth_smallest = 0
        min_heap = []

        def push(i, j):
            if len(matrix) > len(matrix[0]):
                if i < len(matrix[0]) and j < len(matrix):
                    heappush(min_heap, [matrix[j][i], i, j])
            else:
                if i < len(matrix) and j < len(matrix[0]):
                    heappush(min_heap, [matrix[i][j], i, j])

        push(0, 0)
        while min_heap and k > 0:
            kth_smallest, i, j = heappop(min_heap)
            push(i, j + 1)
            if j == 0:
                push(i + 1, 0)
            k -= 1

        return kth_smallest
```

word-ladder.py

```python
# Given two words (beginWord and endWord), and a dictionary's word list, find
# the length of shortest transformation sequence from beginWord to endWord, such
# that:
#
#
#       Only one letter can be changed at a time.
#       Each transformed word must exist in the word list.
#
#
# Note:
#
#
#       Return 0 if there is no such transformation sequence.
#       All words have the same length.
#       All words contain only lowercase alphabetic characters.
#       You may assume no duplicates in the word list.
#       You may assume beginWord and endWord are non-empty and are not the same.
#
#
# Example 1:
#
# Input:
# beginWord = "hit",
# endWord = "cog",
# wordList = ["hot","dot","dog","lot","log","cog"]
#
# Output: 5
#
# Explanation: As one shortest transformation is "hit" -> "hot" -> "dot" ->
# "dog" -> "cog",
# return its length 5.
#
#
# Example 2:
#
# Input:
# beginWord = "hit"
# endWord = "cog"
# wordList = ["hot","dot","dog","lot","log"]
#
# Output: 0
#
# Explanation: The endWord "cog" is not in wordList, therefore no
# possible transformation.# Time:  O(n * d), n is length of string, d is size of dictionary
# Space: O(d)
from string import ascii_lowercase


class Solution(object):
    def ladderLength(self, beginWord, endWord, wordList):
        """
        :type beginWord: str
        :type endWord: str
        :type wordList: List[str]
        :rtype: int
        """
        distance, cur, visited, lookup = 0, [beginWord], set([beginWord]), set(wordList)
```

```
while cur:
    next_queue = []

    for word in cur:
        if word == endWord:
            return distance + 1
        for i in xrange(len(word)):
            for j in ascii_lowercase:
                candidate = word[:i] + j + word[i+1:]
                if candidate not in visited and candidate in lookup:
                    next_queue.append(candidate)
                    visited.add(candidate)
    distance += 1
    cur = next_queue

return 0
```

camelcase-matching.py

```python
# A query word matches a given pattern if we can insert lowercase letters to the
# pattern word so that it equals the query. (We may insert each character at any
# position, and may insert 0 characters.)
#
# Given a list of queries, and a pattern, return an answer list of booleans,
# where answer[i] is true if and only if queries[i] matches the pattern.
#
#
#
# Example 1:
#
# Input: queries =
# ["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern = "FB"
# Output: [true,false,true,true,false]
# Explanation:
# "FooBar" can be generated like this "F" + "oo" + "B" + "ar".
# "FootBall" can be generated like this "F" + "oot" + "B" + "all".
# "FrameBuffer" can be generated like this "F" + "rame" + "B" + "uffer".
#
# Example 2:
#
# Input: queries =
# ["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern =
# "FoBa"
# Output: [true,false,true,false,false]
# Explanation:
# "FooBar" can be generated like this "Fo" + "o" + "Ba" + "r".
# "FootBall" can be generated like this "Fo" + "ot" + "Ba" + "ll".
#
#
# Example 3:
#
# Input: queries =
# ["FooBar","FooBarTest","FootBall","FrameBuffer","ForceFeedBack"], pattern =
# "FoBaT"
# Output: [false,true,false,false,false]
# Explanation:
# "FooBarTest" can be generated like this "Fo" + "o" + "Ba" + "r" + "T" + "est".
#
#
#
#
# Note:
#
#
#       1 <= queries.length <= 100
#       1 <= queries[i].length <= 100
#       1 <= pattern.length <= 100
#       All strings consists only of lower and upper case English letters.# Time:  O(n * l), n is number of qu
#                , l is length of query
# Space: O(1)

class Solution(object):
    def camelMatch(self, queries, pattern):
        """
        :type queries: List[str]
        :type pattern: str
        :rtype: List[bool]
```

```python
"""
def is_matched(query, pattern):
    i = 0
    for c in query:
        if i < len(pattern) and pattern[i] == c:
            i += 1
        elif c.isupper():
            return False
    return i == len(pattern)

result = []
for query in queries:
    result.append(is_matched(query, pattern))
return result
```

encode-and-decode-tinyurl.py

```python
# Note: This is a companion problem to the System Design problem: Design
# TinyURL.
#
# TinyURL is a URL shortening service where you enter a URL such as
# https://leetcode.com/problems/design-tinyurl and it returns a short URL such as
# http://tinyurl.com/4e9iAk.
#
# Design the encode and decode methods for the TinyURL service. There is no
# restriction on how your encode/decode algorithm should work. You just need to
# ensure that a URL can be encoded to a tiny URL and the tiny URL can be decoded
# to the original URL.# Time:  O(1)
# Space: O(n)

import random


class Codec(object):
    def __init__(self):
        self.__random_length = 6
        self.__tiny_url = "http://tinyurl.com/"
        self.__alphabet = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
        self.__lookup = {}

    def encode(self, longUrl):
        """Encodes a URL to a shortened URL.

        :type longUrl: str
        :rtype: str
        """
        def getRand():
            rand = []
            for _ in xrange(self.__random_length):
                rand += self.__alphabet[random.randint(0, len(self.__alphabet)-1)]
            return "".join(rand)

        key = getRand()
        while key in self.__lookup:
            key = getRand()
        self.__lookup[key] = longUrl
        return self.__tiny_url + key

    def decode(self, shortUrl):
        """Decodes a shortened URL to its original URL.

        :type shortUrl: str
        :rtype: str
        """
        return self.__lookup[shortUrl[len(self.__tiny_url):]]


from hashlib import sha256


class Codec2(object):

    def __init__(self):
        self._cache = {}
        self.url = 'http://tinyurl.com/'
```

```python
def encode(self, long_url):
    """Encodes a URL to a shortened URL.

    :type long_url: str
    :rtype: str
    """
    key = sha256(long_url.encode()).hexdigest()[:6]
    self._cache[key] = long_url
    return self.url + key

def decode(self, short_url):
    """Decodes a shortened URL to its original URL.

    :type short_url: str
    :rtype: str
    """
    key = short_url.replace(self.url, '')
    return self._cache[key]
```

triangle.py

```python
# 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.from functools import reduce
# Time:  O(m * n)
# Space: O(n)

class Solution(object):
    # @param triangle, a list of lists of integers
    # @return an integer
    def minimumTotal(self, triangle):
        if not triangle:
            return 0

        cur = triangle[0] + [float("inf")]
        for i in xrange(1, len(triangle)):
            next = []
            next.append(triangle[i][0] + cur[0])
            for j in xrange(1, i + 1):
                next.append(triangle[i][j] + min(cur[j - 1], cur[j]))
            cur = next + [float("inf")]

        return reduce(min, cur)
```

get-equal-substrings-within-budget.py

```
# You are given two strings s and t of the same length. You want to change s to
# t. Changing the i-th character of s to i-th character of t costs |s[i] - t[i]|
# that is, the absolute difference between the ASCII values of the characters.
#
# You are also given an integer maxCost.
#
# Return the maximum length of a substring of s that can be changed to be the
# same as the corresponding substring of twith a cost less than or equal to
# maxCost.
#
# If there is no substring from s that can be changed to its corresponding
# substring from t, return 0.
#
#
# Example 1:
#
# Input: s = "abcd", t = "bcdf", maxCost = 3
# Output: 3
# Explanation: "abc" of s can change to "bcd". That costs 3, so the maximum
# length is 3.
#
# Example 2:
#
# Input: s = "abcd", t = "cdef", maxCost = 3
# Output: 1
# Explanation: Each character in s costs 2 to change to charactor in t, so the
# maximum length is 1.
#
#
# Example 3:
#
# Input: s = "abcd", t = "acde", maxCost = 0
# Output: 1
# Explanation: You can't make any change, so the maximum length is 1.
#
#
#
# Constraints:
#
#
#       1 <= s.length, t.length <= 10^5
#       0 <= maxCost <= 10^6
#       s and t only contain lower case English letters.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def equalSubstring(self, s, t, maxCost):
        """
        :type s: str
        :type t: str
        :type maxCost: int
        :rtype: int
        """
        left = 0
        for right in xrange(len(s)):
            maxCost -= abs(ord(s[right])-ord(t[right]))
            if maxCost < 0:
                maxCost += abs(ord(s[left])-ord(t[left]))
```

```
        left += 1
return (right+1)-left
```

score-after-flipping-matrix.py

```python
# We have a two dimensional matrix A where each value is 0 or 1.
#
# A move consists of choosing any row or column, and toggling each value in that
# row or column: changing all 0s to 1s, and all 1s to 0s.
#
# After making any number of moves, every row of this matrix is interpreted as a
# binary number, and the score of the matrix is the sum of these numbers.
#
# Return the highest possible score.
#
#
#
#
#
#
#
# Example 1:
#
# Input: [[0,0,1,1],[1,0,1,0],[1,1,0,0]]
# Output: 39
# Explanation:
# Toggled to [[1,1,1,1],[1,0,0,1],[1,1,1,1]].
# 0b1111 + 0b1001 + 0b1111 = 15 + 9 + 15 = 39
#
#
#
# Note:
#
#
#       1 <= A.length <= 20
#       1 <= A[0].length <= 20
#       A[i][j] is 0 or 1.# Time:  O(r * c)
# Space: O(1)


class Solution(object):
    def matrixScore(self, A):
        """
        :type A: List[List[int]]
        :rtype: int
        """
        R, C = len(A), len(A[0])
        result = 0
        for c in xrange(C):
            col = 0
            for r in xrange(R):
                col += A[r][c] ^ A[r][0]
            result += max(col, R-col) * 2**(C-1-c)
        return result
```

subarray-sums-divisible-by-k.py

```python
# Given an array A of integers, return the number of (contiguous, non-empty)
# subarrays that have a sum divisible by K.
#
#
#
#
# Example 1:
#
# Input: A = [4,5,0,-2,-3,1], K = 5
# Output: 7
# Explanation: There are 7 subarrays with a sum divisible by K = 5:
# [4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]
#
#
#
#
# Note:
#
#
#       1 <= A.length <= 30000
#       -10000 <= A[i] <= 10000
#       2 <= K <= 10000# Time:  O(n)
# Space: O(k)

import collections


class Solution(object):
    def subarraysDivByK(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        count = collections.defaultdict(int)
        count[0] = 1
        result, prefix = 0, 0
        for a in A:
            prefix = (prefix+a) % K
            result += count[prefix]
            count[prefix] += 1
        return result
```

lowest-common-ancestor-of-deepest-leaves.py

```python
# Given a rooted binary tree, return the lowest common ancestor of its deepest
# leaves.
#
# Recall that:
#
#
#       The node of a binary tree is a leaf if and only if it has no children
#       The depth of the root of the tree is 0, and if the depth of a node is d,
# the depth of each of its children is d+1.
#       The lowest common ancestor of a set S of nodes is the node A with the
# largest depth such that every node in S is in the subtree with root A.
#
#
#
# Example 1:
#
# Input: root = [1,2,3]
# Output: [1,2,3]
# Explanation:
# The deepest leaves are the nodes with values 2 and 3.
# The lowest common ancestor of these leaves is the node with value 1.
# The answer returned is a TreeNode object (not an array) with serialization
# "[1,2,3]".
#
#
# Example 2:
#
# Input: root = [1,2,3,4]
# Output: [4]
#
#
# Example 3:
#
# Input: root = [1,2,3,4,5]
# Output: [2,4,5]
#
#
#
# Constraints:
#
#
#       The given tree will have between 1 and 1000 nodes.
#       Each node of the tree will have a distinct value between 1 and 1000.# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def lcaDeepestLeaves(self, root):
        """
        :type root: TreeNode
        :rtype: TreeNode
```

```python
"""
def lcaDeepestLeavesHelper(root):
    if not root:
        return 0, None
    d1, lca1 = lcaDeepestLeavesHelper(root.left)
    d2, lca2 = lcaDeepestLeavesHelper(root.right)
    if d1 > d2:
        return d1+1, lca1
    if d1 < d2:
        return d2+1, lca2
    return d1+1, root

return lcaDeepestLeavesHelper(root)[1]
```

3sum-with-multiplicity.py

```python
# Given an integer array A, and an integer target, return the number
# of tuples i, j, k  such that i < j < k and A[i] + A[j] + A[k] == target.
#
# As the answer can be very large, return it modulo 109 + 7.
#
#
# Example 1:
#
# Input: A = [1,1,2,2,3,3,4,4,5,5], target = 8
# Output: 20
# Explanation:
# Enumerating by the values (A[i], A[j], A[k]):
# (1, 2, 5) occurs 8 times;
# (1, 3, 4) occurs 8 times;
# (2, 2, 4) occurs 2 times;
# (2, 3, 3) occurs 2 times.
#
#
# Example 2:
#
# Input: A = [1,1,2,2,2,2], target = 5
# Output: 12
# Explanation:
# A[i] = 1, A[j] = A[k] = 2 occurs 12 times:
# We choose one 1 from [1,1] in 2 ways,
# and two 2s from [2,2,2,2] in 6 ways.
#
#
#
# Constraints:
#
#
#       3 <= A.length <= 3000
#       0 <= A[i] <= 100
#       0 <= target <= 300# Time:  O(n^2), n is the number of disctinct A[i]
# Space: O(n)

import collections
import itertools


class Solution(object):
    def threeSumMulti(self, A, target):
        """
        :type A: List[int]
        :type target: int
        :rtype: int
        """
        count = collections.Counter(A)
        result = 0
        for i, j in itertools.combinations_with_replacement(count, 2):
            k = target - i - j
            if i == j == k:
                result += count[i] * (count[i]-1) * (count[i]-2) // 6
            elif i == j != k:
                result += count[i] * (count[i]-1) // 2 * count[k]
            elif max(i, j) < k:
                result += count[i] * count[j] * count[k]
```

```
    return result % (10**9 + 7)
```

max-consecutive-ones-iii.py

```python
# Given an array A of 0s and 1s, we may change up to K values from 0 to 1.
#
# Return the length of the longest (contiguous) subarray that contains only 1s.
#
#
#
#
# Example 1:
#
# Input: A = [1,1,1,0,0,0,1,1,1,1,0], K = 2
# Output: 6
# Explanation:
# [1,1,1,0,0,1,1,1,1,1,1]
# Bolded numbers were flipped from 0 to 1.  The longest subarray is underlined.
#
#
# Example 2:
#
# Input: A = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], K = 3
# Output: 10
# Explanation:
# [0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1]
# Bolded numbers were flipped from 0 to 1.  The longest subarray is underlined.
#
#
#
#
# Note:
#
#
#       1 <= A.length <= 20000
#       0 <= K <= A.length
#       A[i] is 0 or 1# Time:  O(n)
# Space: O(1)

class Solution(object):
    def longestOnes(self, A, K):
        """
        :type A: List[int]
        :type K: int
        :rtype: int
        """
        result, i = 0, 0
        for j in xrange(len(A)):
            K -= int(A[j] == 0)
            while K < 0:
                K += int(A[i] == 0)
                i += 1
            result = max(result, j-i+1)
        return result
```

adding-two-negabinary-numbers.py

```python
# Given two numbers arr1 and arr2 in base -2, return the result of adding them
# together.
#
# Each number is given in array format:  as an array of 0s and 1s, from most
# significant bit to least significant bit.  For example, arr = [1,1,0,1]
# represents the number (-2)^3 + (-2)^2 + (-2)^0 = -3.  A number arr in array
# format is also guaranteed to have no leading zeros: either arr == [0] or arr[0]
# == 1.
#
# Return the result of adding arr1 and arr2 in the same format: as an array of
# 0s and 1s with no leading zeros.
#
#
#
# Example 1:
#
# Input: arr1 = [1,1,1,1,1], arr2 = [1,0,1]
# Output: [1,0,0,0,0]
# Explanation: arr1 represents 11, arr2 represents 5, the output represents 16.
#
#
#
#
# Note:
#
#
#       1 <= arr1.length <= 1000
#       1 <= arr2.length <= 1000
#       arr1 and arr2 have no leading zeros
#       arr1[i] is 0 or 1
#       arr2[i] is 0 or 1# Time:  O(n)
# Space: O(n)

class Solution(object):
    def addNegabinary(self, arr1, arr2):
        """
        :type arr1: List[int]
        :type arr2: List[int]
        :rtype: List[int]
        """
        result = []
        carry = 0
        while arr1 or arr2 or carry:
            if arr1:
                carry += arr1.pop()
            if arr2:
                carry += arr2.pop()
            result.append(carry & 1)
            carry = -(carry >> 1)
        while len(result) > 1 and result[-1] == 0:
            result.pop()
        result.reverse()
        return result
```

map-sum-pairs.py

```python
# Implement a MapSum class with insert, and sum methods.
#
#
#
# For the method insert, you'll be given a pair of (string, integer). The string
# represents the key and the integer represents the value. If the key already
# existed, then the original key-value pair will be overridden to the new one.
#
#
#
# For the method sum, you'll be given a string representing the prefix, and you
# need to return the sum of all the pairs' value whose key starts with the prefix.
#
#
# Example 1:
#
# Input: insert("apple", 3), Output: Null
# Input: sum("ap"), Output: 3
# Input: insert("app", 2), Output: Null
# Input: sum("ap"), Output: 5# Time:  O(n), n is the length of key
# Space: O(t), t is the number of nodes in trie

import collections


class MapSum(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        _trie = lambda: collections.defaultdict(_trie)
        self.__root = _trie()


    def insert(self, key, val):
        """
        :type key: str
        :type val: int
        :rtype: void
        """
        # Time: O(n)
        curr = self.__root
        for c in key:
            curr = curr[c]
        delta = val
        if "_end" in curr:
            delta -= curr["_end"]

        curr = self.__root
        for c in key:
            curr = curr[c]
            if "_count" in curr:
                curr["_count"] += delta
            else:
                curr["_count"] = delta
        curr["_end"] = val
```

```python
def sum(self, prefix):
    """
    :type prefix: str
    :rtype: int
    """
    # Time: O(n)
    curr = self.__root
    for c in prefix:
        if c not in curr:
            return 0
        curr = curr[c]
    return curr["_count"]
```

binary-tree-coloring-game.py

```python
# Two players play a turn based game on a binary tree.  We are given the root of
# this binary tree, and the number of nodes n in the tree.  n is odd, and each
# node has a distinct value from 1 to n.
#
# Initially, the first player names a value x with 1 <= x <= n, and the second
# player names a value y with 1 <= y <= n and y != x.  The first player colors the
# node with value x red, and the second player colors the node with value y blue.
#
# Then, the players take turns starting with the first player.  In each turn,
# that player chooses a node of their color (red if player 1, blue if player 2)
# and colors an uncolored neighbor of the chosen node (either the left child,
# right child, or parent of the chosen node.)
#
# If (and only if) a player cannot choose such a node in this way, they must
# pass their turn.  If both players pass their turn, the game ends, and the winner
# is the player that colored more nodes.
#
# You are the second player.  If it is possible to choose such a y to ensure you
# win the game, return true.  If it is not possible, return false.
#
#
# Example 1:
#
# Input: root = [1,2,3,4,5,6,7,8,9,10,11], n = 11, x = 3
# Output: true
# Explanation: The second player can choose the node with value 2.
#
#
#
# Constraints:
#
#
#       root is the root of a binary tree with n nodes and distinct node values
# from 1 to n.
#       n is odd.
#       1 <= x <= n <= 100# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def btreeGameWinningMove(self, root, n, x):
        """
        :type root: TreeNode
        :type n: int
        :type x: int
        :rtype: bool
        """
        def count(node, x, left_right):
            if not node:
                return 0
            left, right = count(node.left, x, left_right), count(node.right, x, left_right)
```

```
    if node.val == x:
        left_right[0], left_right[1] = left, right
    return left + right + 1

left_right = [0, 0]
count(root, x, left_right)
blue = max(max(left_right), n-(sum(left_right)+1))
return blue > n-blue
```

non-overlapping-intervals.py

```python
# Given a collection of intervals, find the minimum number of intervals you need
# to remove to make the rest of the intervals non-overlapping.
#
#
#
#
#
#
# Example 1:
#
# Input: [[1,2],[2,3],[3,4],[1,3]]
# Output: 1
# Explanation: [1,3] can be removed and the rest of intervals are non-
# overlapping.
#
#
# Example 2:
#
# Input: [[1,2],[1,2],[1,2]]
# Output: 2
# Explanation: You need to remove two [1,2] to make the rest of intervals non-
# overlapping.
#
#
# Example 3:
#
# Input: [[1,2],[2,3]]
# Output: 0
# Explanation: You don't need to remove any of the intervals since they're
# already non-overlapping.
#
#
#
#
# Note:
#
#
#       You may assume the interval's end point is always bigger than its start
# point.
#       Intervals like [1,2] and [2,3] have borders "touching" but they don't
# overlap each other.# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def eraseOverlapIntervals(self, intervals):
        """
        :type intervals: List[Interval]
        :rtype: int
        """
        intervals.sort(key=lambda interval: interval.start)
        result, prev = 0, 0
        for i in xrange(1, len(intervals)):
            if intervals[i].start < intervals[prev].end:
                if intervals[i].end < intervals[prev].end:
                    prev = i
                result += 1
            else:
                prev = i
```

```
    return result
```

reverse-substrings-between-each-pair-of-parentheses.py

```python
# You are given a string s that consists of lower case English letters and
# brackets.
#
# Reverse the strings in each pair of matching parentheses, starting from the
# innermost one.
#
# Your result should not contain any brackets.
#
#
# Example 1:
#
# Input: s = "(abcd)"
# Output: "dcba"
#
#
# Example 2:
#
# Input: s = "(u(love)i)"
# Output: "iloveu"
# Explanation: The substring "love" is reversed first, then the whole string is
# reversed.
#
#
# Example 3:
#
# Input: s = "(ed(et(oc))el)"
# Output: "leetcode"
# Explanation: First, we reverse the substring "oc", then "etco", and finally,
# the whole string.
#
#
# Example 4:
#
# Input: s = "a(bcdefghijkl(mno)p)q"
# Output: "apmnolkjihgfedcbq"
#
#
#
# Constraints:
#
#
#       0 <= s.length <= 2000
#       s only contains lower case English characters and parentheses.
#       It's guaranteed that all parentheses are balanced.# Time:  O(n)
# Space: O(n)

class Solution(object):
    def reverseParentheses(self, s):
        """
        :type s: str
        :rtype: str
        """
        stk, lookup = [], {}
        for i, c in enumerate(s):
            if c == '(':
                stk.append(i)
            elif c == ')':
                j = stk.pop()
```

```python
                lookup[i], lookup[j] = j, i
        result = []
        i, d = 0, 1
        while i < len(s):
            if i in lookup:
                i = lookup[i]
                d *= -1
            else:
                result.append(s[i])
            i += d
        return "".join(result)


# Time:  O(n^2)
# Space: O(n)
class Solution2(object):
    def reverseParentheses(self, s):
        """
        :type s: str
        :rtype: str
        """
        stk = [[]]
        for c in s:
            if c == '(':
                stk.append([])
            elif c == ')':
                end = stk.pop()
                end.reverse()
                stk[-1].extend(end)
            else:
                stk[-1].append(c)
        return "".join(stk.pop())
```

restore-ip-addresses.py

```
# Given a string s containing only digits. Return all possible valid IP
# addresses that can be obtained from s. You can return them in any order.
#
# A valid IP address consists of exactly four integers, each integer is between
# 0 and 255, separated by single points and cannot have leading zeros. For
# example, "0.1.2.201" and "192.168.1.1" are valid IP addresses and
# "0.011.255.245", "192.168.1.312" and "192.168@1.1" are invalid IP addresses.
#
#
# Example 1:
# Input: s = "25525511135"
# Output: ["255.255.11.135","255.255.111.35"]
# Example 2:
# Input: s = "0000"
# Output: ["0.0.0.0"]
# Example 3:
# Input: s = "1111"
# Output: ["1.1.1.1"]
# Example 4:
# Input: s = "010010"
# Output: ["0.10.0.10","0.100.1.0"]
# Example 5:
# Input: s = "101023"
# Output: ["1.0.10.23","1.0.102.3","10.1.0.23","10.10.2.3","101.0.2.3"]
#
#
# Constraints:
#
#
#       0 <= s.length <= 3000
#       s consists of digits only.# Time:  O(n^m) = O(3^4)
# Space: O(n * m) = O(3 * 4)

class Solution(object):
    # @param s, a string
    # @return a list of strings
    def restoreIpAddresses(self, s):
        result = []
        self.restoreIpAddressesRecur(result, s, 0, "", 0)
        return result

    def restoreIpAddressesRecur(self, result, s, start, current, dots):
        # pruning to improve performance
        if (4 - dots) * 3 < len(s) - start or (4 - dots) > len(s) - start:
            return

        if start == len(s) and dots == 4:
            result.append(current[:-1])
        else:
            for i in xrange(start, start + 3):
                if len(s) > i and self.isValid(s[start:i + 1]):
                    current += s[start:i + 1] + '.'
                    self.restoreIpAddressesRecur(result, s, i + 1, current, dots + 1)
                    current = current[:-(i - start + 2)]

    def isValid(self, s):
        if len(s) == 0 or (s[0] == '0' and s != "0"):
            return False
```

```
return int(s) < 256
```

random-point-in-non-overlapping-rectangles.py

```python
# Example 1:
#
# Input:
# ["Solution","pick","pick","pick"]
# [[[[1,1,5,5]]],[],[],[]]
# Output:
# [null,[4,1],[4,1],[3,3]]
#
#
#
# Example 2:
#
# Input:
# ["Solution","pick","pick","pick","pick","pick"]
# [[[[-2,-2,-1,-1],[1,0,3,0]]],[],[],[],[],[]]
# Output:
# [null,[-1,-2],[2,0],[-2,-1],[3,0],[-2,-2]]
#
#
#
# Explanation of Input Syntax:
#
# The input is two lists: the subroutines called and
# their arguments. Solution's constructor has one argument, the array of
# rectangles rects. pick has no arguments. Arguments are always wrapped with a
# list, even if there aren't any.# Time:  ctor: O(n)
#        pick: O(logn)
# Space: O(n)

import random
import bisect


class Solution(object):

    def __init__(self, rects):
        """
        :type rects: List[List[int]]
        """
        self.__rects = list(rects)
        self.__prefix_sum = map(lambda x : (x[2]-x[0]+1)*(x[3]-x[1]+1), rects)
        for i in xrange(1, len(self.__prefix_sum)):
            self.__prefix_sum[i] += self.__prefix_sum[i-1]

    def pick(self):
        """
        :rtype: List[int]
        """
        target = random.randint(0, self.__prefix_sum[-1]-1)
        left = bisect.bisect_right(self.__prefix_sum, target)
        rect = self.__rects[left]
        width, height = rect[2]-rect[0]+1, rect[3]-rect[1]+1
        base = self.__prefix_sum[left]-width*height
        return [rect[0]+(target-base)%width, rect[1]+(target-base)//width]
```

super-ugly-number.py

```
# 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.
#
# Example:
#
# Input: n = 12, primes = [2,7,13,19]
# Output: 32
# Explanation: [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 is a super ugly number for any given primes.
#       The given numbers in primes are in ascending order.
#       0 < k  100, 0 < n  106, 0 < primes[i] < 1000.
#       The nth super ugly number is guaranteed to fit in a 32-bit signed
# integer.# Time:  O(n * k)
# Space: O(n + k)

import heapq


# Heap solution. (620ms)
class Solution(object):
    def nthSuperUglyNumber(self, n, primes):
        """
        :type n: int
        :type primes: List[int]
        :rtype: int
        """
        heap, uglies, idx, ugly_by_last_prime = [], [0] * n, [0] * len(primes), [0] * n
        uglies[0] = 1

        for k, p in enumerate(primes):
            heapq.heappush(heap, (p, k))

        for i in xrange(1, n):
            uglies[i], k = heapq.heappop(heap)
            ugly_by_last_prime[i] = k
            idx[k] += 1
            while ugly_by_last_prime[idx[k]] > k:
                idx[k] += 1
            heapq.heappush(heap, (primes[k] * uglies[idx[k]], k))

        return uglies[-1]

# Time:  O(n * k)
# Space: O(n + k)
# Hash solution. (932ms)
class Solution2(object):
    def nthSuperUglyNumber(self, n, primes):
        """
        :type n: int
        :type primes: List[int]
        :rtype: int
```

```python
        """
        uglies, idx, heap, ugly_set = [0] * n, [0] * len(primes), [], set([1])
        uglies[0] = 1

        for k, p in enumerate(primes):
            heapq.heappush(heap, (p, k))
            ugly_set.add(p)

        for i in xrange(1, n):
            uglies[i], k = heapq.heappop(heap)
            while (primes[k] * uglies[idx[k]]) in ugly_set:
                idx[k] += 1
            heapq.heappush(heap, (primes[k] * uglies[idx[k]], k))
            ugly_set.add(primes[k] * uglies[idx[k]])

        return uglies[-1]

# Time:  O(n * logk) ~ O(n * klogk)
# Space: O(n + k)
class Solution3(object):
    def nthSuperUglyNumber(self, n, primes):
        """
        :type n: int
        :type primes: List[int]
        :rtype: int
        """
        uglies, idx, heap = [1], [0] * len(primes), []
        for k, p in enumerate(primes):
            heapq.heappush(heap, (p, k))

        for i in xrange(1, n):
            min_val, k = heap[0]
            uglies += [min_val]

            while heap[0][0] == min_val:  # worst time: O(klogk)
                min_val, k = heapq.heappop(heap)
                idx[k] += 1
                heapq.heappush(heap, (primes[k] * uglies[idx[k]], k))

        return uglies[-1]

# Time:  O(n * k)
# Space: O(n + k)
# TLE due to the last test case, but it passess and performs the best in C++.
class Solution4(object):
    def nthSuperUglyNumber(self, n, primes):
        """
        :type n: int
        :type primes: List[int]
        :rtype: int
        """
        uglies = [0] * n
        uglies[0] = 1
        ugly_by_prime = list(primes)
        idx = [0] * len(primes)

        for i in xrange(1, n):
            uglies[i] = min(ugly_by_prime)
            for k in xrange(len(primes)):
                if uglies[i] == ugly_by_prime[k]:
```

```python
                idx[k] += 1
                ugly_by_prime[k] = primes[k] * uglies[idx[k]]

        return uglies[-1]

# Time:  O(n * logk) ~ O(n * klogk)
# Space: O(k^2)
# TLE due to the last test case, but it passess and performs well in C++.
class Solution5(object):
    def nthSuperUglyNumber(self, n, primes):
        """
        :type n: int
        :type primes: List[int]
        :rtype: int
        """
        ugly_number = 0

        heap = []
        heapq.heappush(heap, 1)
        for p in primes:
            heapq.heappush(heap, p)
        for _ in xrange(n):
            ugly_number = heapq.heappop(heap)
            for i in xrange(len(primes)):
                if ugly_number % primes[i] == 0:
                    for j in xrange(i + 1):
                        heapq.heappush(heap, ugly_number * primes[j])
                    break

        return ugly_number
```

minimum-increment-to-make-array-unique.py

```
# Given an array of integers A, a move consists of choosing any A[i], and
# incrementing it by 1.
#
# Return the least number of moves to make every value in A unique.
#
#
#
# Example 1:
#
# Input: [1,2,2]
# Output: 1
# Explanation:  After 1 move, the array could be [1, 2, 3].
#
#
#
# Example 2:
#
# Input: [3,2,1,2,1,7]
# Output: 6
# Explanation:  After 6 moves, the array could be [3, 4, 1, 2, 5, 7].
# It can be shown with 5 or less moves that it is impossible for the array to
# have all unique values.
#
#
#
#
#
# Note:
#
#
#       0 <= A.length <= 40000
#       0 <= A[i] < 40000# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def minIncrementForUnique(self, A):
        """
        :type A: List[int]
        :rtype: int
        """
        A.sort()
        A.append(float("inf"))
        result, duplicate = 0, 0
        for i in xrange(1, len(A)):
            if A[i-1] == A[i]:
                duplicate += 1
                result -= A[i]
            else:
                move = min(duplicate, A[i]-A[i-1]-1)
                duplicate -= move
                result += move*A[i-1] + move*(move+1)//2
        return result
```

find-right-interval.py

```
# Given a set of intervals, for each of the interval i, check if there exists an
# interval j whose start point is bigger than or equal to the end point of the
# interval i, which can be called that j is on the "right" of i.
#
# For any interval i, you need to store the minimum interval j's index, which
# means that the interval j has the minimum start point to build the "right"
# relationship for interval i. If the interval j doesn't exist, store -1 for the
# interval i. Finally, you need output the stored value of each interval as an
# array.
#
# Note:
#
#
#       You may assume the interval's end point is always bigger than its start
# point.
#       You may assume none of these intervals have the same start point.
#
#
#
#
# Example 1:
#
# Input: [ [1,2] ]
#
# Output: [-1]
#
# Explanation: There is only one interval in the collection, so it outputs -1.
#
#
#
#
# Example 2:
#
# Input: [ [3,4], [2,3], [1,2] ]
#
# Output: [-1, 0, 1]
#
# Explanation: There is no satisfied "right" interval for [3,4].
# For [2,3], the interval [3,4] has minimum-"right" start point;
# For [1,2], the interval [2,3] has minimum-"right" start point.
#
#
#
#
# Example 3:
#
# Input: [ [1,4], [2,3], [3,4] ]
#
# Output: [-1, 2, -1]
#
# Explanation: There is no satisfied "right" interval for [1,4] and [3,4].
# For [2,3], the interval [3,4] has minimum-"right" start point.
#
#
# NOTE: input types have been changed on April 15, 2019. Please reset to default
# code definition to get new method signature.# Time:  O(nlogn)
# Space: O(n)
```

```python
import bisect


class Solution(object):
    def findRightInterval(self, intervals):
        """
        :type intervals: List[Interval]
        :rtype: List[int]
        """
        sorted_intervals = sorted((interval.start, i) for i, interval in enumerate(intervals))
        result = []
        for interval in intervals:
            idx = bisect.bisect_left(sorted_intervals, (interval.end,))
            result.append(sorted_intervals[idx][1] if idx < len(sorted_intervals) else -1)
        return result
```

longest-happy-string.py

```
# A string is called happy if it does not have any of the strings 'aaa',
# 'bbb' or 'ccc' as a substring.
#
# Given three integers a, b and c, return any string s, which satisfies
# following conditions:
#
#
#       s is happy and longest possible.
#       s contains at most a occurrences of the letter 'a', at most
# b occurrences of the letter 'b' and at most c occurrences of the letter 'c'.
#       s will only contain 'a', 'b' and 'c' letters.
#
#
# If there is no such string s return the empty string "".
#
#
# Example 1:
#
# Input: a = 1, b = 1, c = 7
# Output: "ccaccbcc"
# Explanation: "ccbccacc" would also be a correct answer.
#
#
# Example 2:
#
# Input: a = 2, b = 2, c = 1
# Output: "aabbc"
#
#
# Example 3:
#
# Input: a = 7, b = 1, c = 0
# Output: "aabaa"
# Explanation: It's the only correct answer in this case.
#
#
#
# Constraints:
#
#
#       0 <= a, b, c <= 100
#       a + b + c > 0# Time:  O(n)
# Space: O(1)

import heapq


class Solution(object):
    def longestDiverseString(self, a, b, c):
        """
        :type a: int
        :type b: int
        :type c: int
        :rtype: str
        """
        max_heap = []
        if a:
            heapq.heappush(max_heap, (-a, 'a'))
```

```python
        if b:
            heapq.heappush(max_heap, (-b, 'b'))
        if c:
            heapq.heappush(max_heap, (-c, 'c'))
        result = []
        while max_heap:
            count1, c1 = heapq.heappop(max_heap)
            if len(result) >= 2 and result[-1] == result[-2] == c1:
                if not max_heap:
                    return "".join(result)
                count2, c2 = heapq.heappop(max_heap)
                result.append(c2)
                count2 += 1
                if count2:
                    heapq.heappush(max_heap, (count2, c2))
                heapq.heappush(max_heap, (count1, c1))
                continue
            result.append(c1)
            count1 += 1
            if count1 != 0:
                heapq.heappush(max_heap, (count1, c1))
        return "".join(result)


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def longestDiverseString(self, a, b, c):
        """
        :type a: int
        :type b: int
        :type c: int
        :rtype: str
        """
        choices = [[a, 'a'], [b, 'b'], [c, 'c']]
        result = []
        for _ in xrange(a+b+c):
            choices.sort(reverse=True)
            for i, (x, c) in enumerate(choices):
                if x and result[-2:] != [c, c]:
                    result.append(c)
                    choices[i][0] -= 1
                    break
            else:
                break
        return "".join(result)
```

generate-parentheses.py

```python
# 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:
#
# [
#   "((()))",
#   "(()())",
#   "(())()",
#   "()(())",
#   "()()()"
# ]# Time:  O(4^n / n^(3/2)) ~= Catalan numbers
# Space: O(n)

class Solution(object):
    # @param an integer
    # @return a list of string
    def generateParenthesis(self, n):
        result = []
        self.generateParenthesisRecu(result, "", n, n)
        return result

    def generateParenthesisRecu(self, result, current, left, right):
        if left == 0 and right == 0:
            result.append(current)
        if left > 0:
            self.generateParenthesisRecu(result, current + "(", left - 1, right)
        if left < right:
            self.generateParenthesisRecu(result, current + ")", left, right - 1)
```

target-sum.py

```python
# You are given a list of non-negative integers, a1, a2, ..., an, and a target,
# S. Now you have 2 symbols + and -. For each integer, you should choose one from
# + and - as its new symbol.
#
# Find out how many ways to assign symbols to make sum of integers equal to
# target S.
#
# Example 1:
#
# Input: nums is [1, 1, 1, 1, 1], S is 3.
# Output: 5
# Explanation:
#
# -1+1+1+1+1 = 3
# +1-1+1+1+1 = 3
# +1+1-1+1+1 = 3
# +1+1+1-1+1 = 3
# +1+1+1+1-1 = 3
#
# There are 5 ways to assign symbols to make the sum of nums be target 3.
#
#
#
# Constraints:
#
#
#       The length of the given array is positive and will not exceed 20.
#       The sum of elements in the given array will not exceed 1000.
#       Your output answer is guaranteed to be fitted in a 32-bit integer.# Time:  O(n * S)
# Space: O(S)

import collections


class Solution(object):
    def findTargetSumWays(self, nums, S):
        """
        :type nums: List[int]
        :type S: int
        :rtype: int
        """
        def subsetSum(nums, S):
            dp = collections.defaultdict(int)
            dp[0] = 1
            for n in nums:
                for i in reversed(xrange(n, S+1)):
                    if i-n in dp:
                        dp[i] += dp[i-n]
            return dp[S]

        total = sum(nums)
        if total < S or (S + total) % 2: return 0
        P = (S + total) // 2
        return subsetSum(nums, P)
```

permutation-in-string.py

```python
# Given two strings s1 and s2, write a function to return true if s2 contains
# the permutation of s1. In other words, one of the first string's permutations is
# the substring of the second string.
#
#
#
# Example 1:
#
# Input: s1 = "ab" s2 = "eidbaooo"
# Output: True
# Explanation: s2 contains one permutation of s1 ("ba").
#
#
# Example 2:
#
# Input:s1= "ab" s2 = "eidboaoo"
# Output: False
#
#
#
# Constraints:
#
#
#       The input strings only contain lower case letters.
#       The length of both given strings is in range [1, 10,000].# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def checkInclusion(self, s1, s2):
        """
        :type s1: str
        :type s2: str
        :rtype: bool
        """
        counts = collections.Counter(s1)
        l = len(s1)
        for i in xrange(len(s2)):
            if counts[s2[i]] > 0:
                l -= 1
            counts[s2[i]] -= 1
            if l == 0:
                return True
            start = i + 1 - len(s1)
            if start >= 0:
                counts[s2[start]] += 1
                if counts[s2[start]] > 0:
                    l += 1
        return False
```

maximum-swap.py

```
# Given a non-negative integer, you could swap two digits at most once to get
# the maximum valued number. Return the maximum valued number you could get.
#
#
# Example 1:
#
# Input: 2736
# Output: 7236
# Explanation: Swap the number 2 and the number 7.
#
#
#
# Example 2:
#
# Input: 9973
# Output: 9973
# Explanation: No swap.
#
#
#
#
# Note:
#
#
# The given number is in the range [0, 108]# Time:  O(logn), logn is the length of the number string
# Space: O(logn)

class Solution(object):
    def maximumSwap(self, num):
        """
        :type num: int
        :rtype: int
        """
        digits = list(str(num))
        left, right = 0, 0
        max_idx = len(digits)-1
        for i in reversed(xrange(len(digits))):
            if digits[i] > digits[max_idx]:
                max_idx = i
            elif digits[max_idx] > digits[i]:
                left, right = i, max_idx
        digits[left], digits[right] = digits[right], digits[left]
        return int("".join(digits))
```

best-time-to-buy-and-sell-stock-with-cooldown.py

```python
# 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:
#
# Input: [1,2,3,0,2]
# Output: 3
# Explanation: transactions = [buy, sell, cooldown, buy, sell]# Time:  O(n)
# Space: O(1)


class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]
        :rtype: int
        """
        if not prices:
            return 0
        buy, sell, coolDown = [0] * 2, [0] * 2, [0] * 2
        buy[0] = -prices[0]
        for i in xrange(1, len(prices)):
            # Bought before or buy today.
            buy[i % 2] = max(buy[(i - 1) % 2],
                             coolDown[(i - 1) % 2] - prices[i])
            # Sell today.
            sell[i % 2] = buy[(i - 1) % 2] + prices[i]
            # Sold before yesterday or sold yesterday.
            coolDown[i % 2] = max(coolDown[(i - 1) % 2], sell[(i - 1) % 2])
        return max(coolDown[(len(prices) - 1) % 2],
                   sell[(len(prices) - 1) % 2])
```

simplify-path.py

```python
# Given an absolute path for a file (Unix-style), simplify it. Or in other
# words, convert it to the canonical path.
#
# In a UNIX-style file system, a period . refers to the current directory.
# Furthermore, a double period .. moves the directory up a level.
#
# Note that the returned canonical path must always begin with a slash /, and
# there must be only a single slash / between two directory names. The last
# directory name (if it exists) must not end with a trailing /. Also, the
# canonical path must be the shortest string representing the absolute path.
#
#
#
# Example 1:
#
# Input: "/home/"
# Output: "/home"
# Explanation: Note that there is no trailing slash after the last directory
# name.
#
#
# Example 2:
#
# Input: "/../"
# Output: "/"
# Explanation: Going one level up from the root directory is a no-op, as the
# root level is the highest level you can go.
#
#
# Example 3:
#
# Input: "/home//foo/"
# Output: "/home/foo"
# Explanation: In the canonical path, multiple consecutive slashes are replaced
# by a single one.
#
#
# Example 4:
#
# Input: "/a/./b/../../c/"
# Output: "/c"
#
#
# Example 5:
#
# Input: "/a/../../b/../c//.//"
# Output: "/c"
#
#
# Example 6:
#
# Input: "/a//b////c/d//././/.."
# Output: "/a/b/c"# Time:  O(n)
# Space: O(n)

class Solution(object):
    # @param path, a string
    # @return a string
```

```python
def simplifyPath(self, path):
    stack, tokens = [], path.split("/")
    for token in tokens:
        if token == ".." and stack:
            stack.pop()
        elif token != ".." and token != "." and token:
            stack.append(token)
    return "/" + "/".join(stack)
```

maximum-binary-tree.py

```python
# Given an integer array with no duplicates. A maximum tree building on this
# array is defined as follow:
#
# The root is the maximum number in the array.
# The left subtree is the maximum tree constructed from left part subarray
# divided by the maximum number.
# The right subtree is the maximum tree constructed from right part subarray
# divided by the maximum number.
#
#
#
#
# Construct the maximum tree by the given array and output the root node of this
# tree.
#
#
# Example 1:
#
# Input: [3,2,1,6,0,5]
# Output: return the tree root node representing the following tree:
#
#       6
#     /   \
#    3     5
#     \    /
#      2  0
#        \
#         1
#
#
#
# Note:
#
#
# The size of the given array will be in the range [1,1000].# Time:  O(n)
# Space: O(n)

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def constructMaximumBinaryTree(self, nums):
        """
        :type nums: List[int]
        :rtype: TreeNode
        """
        # https://github.com/kamyu104/LintCode/blob/master/C++/max-tree.cpp
        nodeStack = []
        for num in nums:
            node = TreeNode(num)
            while nodeStack and num > nodeStack[-1].val:
                node.left = nodeStack.pop()
            if nodeStack:
                nodeStack[-1].right = node
```

```
    nodeStack.append(node)
return nodeStack[0]
```

decode-string.py

```python
# Given an encoded string, return its decoded string.
#
# The encoding rule is: k[encoded_string], where the encoded_string inside the
# square brackets is being repeated exactly k times. Note that k is guaranteed to
# be a positive integer.
#
# You may assume that the input string is always valid; No extra white spaces,
# square brackets are well-formed, etc.
#
# Furthermore, you may assume that the original data does not contain any digits
# and that digits are only for those repeat numbers, k. For example, there won't
# be input like 3a or 2[4].
#
#
# Example 1:
# Input: s = "3[a]2[bc]"
# Output: "aaabcbc"
# Example 2:
# Input: s = "3[a2[c]]"
# Output: "accaccacc"
# Example 3:
# Input: s = "2[abc]3[cd]ef"
# Output: "abcabccdcdcdef"
# Example 4:
# Input: s = "abc3[cd]xyz"
# Output: "abccdcdcdxyz"# Time:  O(n)
# Space: O(n)

class Solution(object):
    def decodeString(self, s):
        """
        :type s: str
        :rtype: str
        """
        curr, nums, strs = [], [], []
        n = 0

        for c in s:
            if c.isdigit():
                n = n * 10 + ord(c) - ord('0')
            elif c == '[':
                nums.append(n)
                n = 0
                strs.append(curr)
                curr = []
            elif c == ']':
                strs[-1].extend(curr * nums.pop())
                curr = strs.pop()
            else:
                curr.append(c)

        return "".join(strs[-1]) if strs else "".join(curr)
```

smallest-string-with-swaps.py

```python
# You are given a string s, and an array of pairs of indices in the
# string pairs where pairs[i] = [a, b] indicates 2 indices(0-indexed) of the
# string.
#
# You can swap the characters at any pair of indices in the given pairs any
# number of times.
#
# Return the lexicographically smallest string that s can be changed to after
# using the swaps.
#
#
# Example 1:
#
# Input: s = "dcab", pairs = [[0,3],[1,2]]
# Output: "bacd"
# Explaination:
# Swap s[0] and s[3], s = "bcad"
# Swap s[1] and s[2], s = "bacd"
#
#
# Example 2:
#
# Input: s = "dcab", pairs = [[0,3],[1,2],[0,2]]
# Output: "abcd"
# Explaination:
# Swap s[0] and s[3], s = "bcad"
# Swap s[0] and s[2], s = "acbd"
# Swap s[1] and s[2], s = "abcd"
#
# Example 3:
#
# Input: s = "cba", pairs = [[0,1],[1,2]]
# Output: "abc"
# Explaination:
# Swap s[0] and s[1], s = "bca"
# Swap s[1] and s[2], s = "bac"
# Swap s[0] and s[1], s = "abc"
#
#
#
# Constraints:
#
#
#       1 <= s.length <= 10^5
#       0 <= pairs.length <= 10^5
#       0 <= pairs[i][0], pairs[i][1] < s.length
#       s only contains lower case English letters.# Time:  O(nlogn)
# Space: O(n)

import collections


class UnionFind(object):
    def __init__(self, n):
        self.set = range(n)

    def find_set(self, x):
        if self.set[x] != x:
```

```python
            self.set[x] = self.find_set(self.set[x])  # path compression.
        return self.set[x]

    def union_set(self, x, y):
        x_root, y_root = map(self.find_set, (x, y))
        if x_root == y_root:
            return False
        self.set[max(x_root, y_root)] = min(x_root, y_root)
        return True


class Solution(object):
    def smallestStringWithSwaps(self, s, pairs):
        """
        :type s: str
        :type pairs: List[List[int]]
        :rtype: str
        """
        union_find = UnionFind(len(s))
        for x,y in pairs:
            union_find.union_set(x, y)
        components = collections.defaultdict(list)
        for i in xrange(len(s)):
            components[union_find.find_set(i)].append(s[i])
        for i in components.iterkeys():
            components[i].sort(reverse=True)
        result = []
        for i in xrange(len(s)):
            result.append(components[union_find.find_set(i)].pop())
        return "".join(result)


# Time:  O(nlogn)
# Space: O(n)
import itertools
class Solution2(object):
    def smallestStringWithSwaps(self, s, pairs):
        """
        :type s: str
        :type pairs: List[List[int]]
        :rtype: str
        """
        def dfs(i, adj, lookup, component):
            lookup.add(i)
            component.append(i)
            for j in adj[i]:
                if j in lookup:
                    continue
                dfs(j, adj, lookup, component)

        adj = collections.defaultdict(list)
        for i, j in pairs:
            adj[i].append(j)
            adj[j].append(i)
        lookup = set()
        result = list(s)
        for i in xrange(len(s)):
            if i in lookup:
                continue
            component = []
```

```
    dfs(i, adj, lookup, component)
    component.sort()
    chars = sorted(result[k] for k in component)
    for comp, char in itertools.izip(component, chars):
        result[comp] = char
return "".join(result)
```

optimal-division.py

```python
# Given a list of positive integers, the adjacent integers will perform the
# float division. For example, [2,3,4] -> 2 / 3 / 4.
#
# However, you can add any number of parenthesis at any position to change the
# priority of operations. You should find out how to add parenthesis to get the
# maximum result, and return the corresponding expression in string format. Your
# expression should NOT contain redundant parenthesis.
#
# Example:
#
# Input: [1000,100,10,2]
# Output: "1000/(100/10/2)"
# Explanation:
# 1000/(100/10/2) = 1000/((100/10)/2) = 200
# However, the bold parenthesis in "1000/((100/10)/2)" are redundant,
# since they don't influence the operation priority. So you should return
# "1000/(100/10/2)".
#
# Other cases:
# 1000/(100/10)/2 = 50
# 1000/(100/(10/2)) = 50
# 1000/100/10/2 = 0.5
# 1000/100/(10/2) = 2
#
#
#
# Note:
#
# The length of the input array is [1, 10].
# Elements in the given array will be in range [2, 1000].
# There is only one optimal division for each test case.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def optimalDivision(self, nums):
        """
        :type nums: List[int]
        :rtype: str
        """
        if len(nums) == 1:
            return str(nums[0])
        if len(nums) == 2:
            return str(nums[0]) + "/" + str(nums[1])
        result = [str(nums[0]) + "/(" + str(nums[1])]
        for i in xrange(2, len(nums)):
            result += "/" + str(nums[i])
        result += ")"
        return "".join(result)
```

remove-all-adjacent-duplicates-in-string-ii.py

```python
# Given a string s, a k duplicate removal consists of choosing k adjacent and
# equal letters from s and removing them causing the left and the right side of
# the deleted substring to concatenate together.
#
# We repeatedly make k duplicate removals on s until we no longer can.
#
# Return the final string after all such duplicate removals have been made.
#
# It is guaranteed that the answer is unique.
#
#
# Example 1:
#
# Input: s = "abcd", k = 2
# Output: "abcd"
# Explanation: There's nothing to delete.
#
# Example 2:
#
# Input: s = "deeedbbcccbdaa", k = 3
# Output: "aa"
# Explanation:
# First delete "eee" and "ccc", get "ddbbbdaa"
# Then delete "bbb", get "dddaa"
# Finally delete "ddd", get "aa"
#
# Example 3:
#
# Input: s = "pbbcggttciiippooaais", k = 2
# Output: "ps"
#
#
#
# Constraints:
#
#
#       1 <= s.length <= 10^5
#       2 <= k <= 10^4
#       s only contains lower case English letters.# Time:  O(n)
# Space: O(n)

class Solution(object):
    def removeDuplicates(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: str
        """
        stk = [['^', 0]]
        for c in s:
            if stk[-1][0] == c:
                stk[-1][1] += 1
                if stk[-1][1] == k:
                    stk.pop()
            else:
                stk.append([c, 1])
        return "".join(c*k for c, k in stk)
```

clone-graph.py

```
# Given a reference of a node in a connected undirected graph.
#
# Return a deep copy (clone) of the graph.
#
# Each node in the graph contains a val (int) and a list (List[Node]) of its
# neighbors.
#
# class Node {
#     public int val;
#     public List<Node> neighbors;
# }
#
#
#
#
# Test case format:
#
# For simplicity sake, each node's value is the same as the node's index
# (1-indexed). For example, the first node with val = 1, the second node with val
# = 2, and so on. The graph is represented in the test case using an adjacency
# list.
#
# Adjacency list is a collection of unordered lists used to represent a finite
# graph. Each list describes the set of neighbors of a node in the graph.
#
# The given node will always be the first node with val = 1. You must return the
# copy of the given node as a reference to the cloned graph.
#
#
# Example 1:
#
# Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
# Output: [[2,4],[1,3],[2,4],[1,3]]
# Explanation: There are 4 nodes in the graph.
# 1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
# 2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
# 3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
# 4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
#
#
# Example 2:
#
# Input: adjList = [[]]
# Output: [[]]
# Explanation: Note that the input contains one empty list. The graph consists
# of only one node with val = 1 and it does not have any neighbors.
#
#
# Example 3:
#
# Input: adjList = []
# Output: []
# Explanation: This an empty graph, it does not have any nodes.
#
#
# Example 4:
#
# Input: adjList = [[2],[1]]
```

```python
# Output: [[2],[1]]
#
#
#
# Constraints:
#
#
#       1 <= Node.val <= 100
#       Node.val is unique for each node.
#       Number of Nodes will not exceed 100.
#       There is no repeated edges and no self-loops in the graph.
#       The Graph is connected and all nodes can be visited starting from the
# given node.# Time:  O(n)
# Space: O(n)

class UndirectedGraphNode(object):
    def __init__(self, x):
        self.label = x
        self.neighbors = []

class Solution(object):
    # @param node, a undirected graph node
    # @return a undirected graph node
    def cloneGraph(self, node):
        if node is None:
            return None
        cloned_node = UndirectedGraphNode(node.label)
        cloned, queue = {node:cloned_node}, [node]

        while queue:
            current = queue.pop()
            for neighbor in current.neighbors:
                if neighbor not in cloned:
                    queue.append(neighbor)
                    cloned_neighbor = UndirectedGraphNode(neighbor.label)
                    cloned[neighbor] = cloned_neighbor
                cloned[current].neighbors.append(cloned[neighbor])
        return cloned[node]
```

wiggle-sort-ii.py

```python
# Given an unsorted array nums, reorder it such that nums[0] < nums[1] > nums[2]
# < nums[3]....
#
# Example 1:
#
# Input: nums = [1, 5, 1, 1, 6, 4]
# Output: One possible answer is [1, 4, 1, 5, 1, 6].
#
# Example 2:
#
# Input: nums = [1, 3, 2, 2, 3, 1]
# Output: 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?# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def wiggleSort(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        nums.sort()
        med = (len(nums) - 1) / 2
        nums[::2], nums[1::2] = nums[med::-1], nums[:med:-1]

# Time:  O(n) ~ O(n^2)
# Space: O(1)
# Tri Partition (aka Dutch National Flag Problem) with virtual index solution. (TLE)
from random import randint
class Solution2(object):
    def wiggleSort(self, nums):
        """
        :type nums: List[int]
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        def findKthLargest(nums, k):
            left, right = 0, len(nums) - 1
            while left <= right:
                pivot_idx = randint(left, right)
                new_pivot_idx = partitionAroundPivot(left, right, pivot_idx, nums)
                if new_pivot_idx == k - 1:
                    return nums[new_pivot_idx]
                elif new_pivot_idx > k - 1:
                    right = new_pivot_idx - 1
                else:  # new_pivot_idx < k - 1.
                    left = new_pivot_idx + 1

        def partitionAroundPivot(left, right, pivot_idx, nums):
            pivot_value = nums[pivot_idx]
            new_pivot_idx = left
            nums[pivot_idx], nums[right] = nums[right], nums[pivot_idx]
```

```
    for i in xrange(left, right):
        if nums[i] > pivot_value:
            nums[i], nums[new_pivot_idx] = nums[new_pivot_idx], nums[i]
            new_pivot_idx += 1
    nums[right], nums[new_pivot_idx] = nums[new_pivot_idx], nums[right]
    return new_pivot_idx

def reversedTriPartitionWithVI(nums, val):
    def idx(i, N):
        return (1 + 2 * (i)) % N

    N = len(nums) / 2 * 2 + 1
    i, j, n = 0, 0, len(nums) - 1
    while j <= n:
        if nums[idx(j, N)] > val:
            nums[idx(i, N)], nums[idx(j, N)] = nums[idx(j, N)], nums[idx(i, N)]
            i += 1
            j += 1
        elif nums[idx(j, N)] < val:
            nums[idx(j, N)], nums[idx(n, N)] = nums[idx(n, N)], nums[idx(j, N)]
            n -= 1
        else:
            j += 1

mid = (len(nums) - 1) / 2
findKthLargest(nums, mid + 1)
reversedTriPartitionWithVI(nums, nums[mid])
```

unique-binary-search-trees-ii.py

```python
# Given an integer n, generate all structurally unique BST's (binary search
# trees) that store values 1 ... n.
#
# Example:
#
# Input: 3
# Output:
# [
#   [1,null,3,2],
#   [3,2,null,1],
#   [3,1,null,null,2],
#   [2,1,3],
#   [1,null,2,null,3]
# ]
# Explanation:
# The above output corresponds to the 5 unique BST's shown below:
#
#    1         3     3      2      1
#     \       /     /      / \      \
#      3     2     1      1   3      2
#     /     /       \                 \
#    2     1         2                 3
#
#
#
# Constraints:
#
#
#       0 <= n <= 8# Time:  O(4^n / n^(3/2)) ~= Catalan numbers
# Space: O(4^n / n^(3/2)) ~= Catalan numbers

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

    def __repr__(self):
        if self:
            serial = []
            queue = [self]

            while queue:
                cur = queue[0]

                if cur:
                    serial.append(cur.val)
                    queue.append(cur.left)
                    queue.append(cur.right)
                else:
                    serial.append("#")

                queue = queue[1:]

            while serial[-1] == "#":
                serial.pop()

            return repr(serial)
```

```python
        else:
            return None

class Solution(object):
    # @return a list of tree node
    def generateTrees(self, n):
        return self.generateTreesRecu(1, n)

    def generateTreesRecu(self, low, high):
        result = []
        if low > high:
            result.append(None)
        for i in xrange(low, high + 1):
            left = self.generateTreesRecu(low, i - 1)
            right = self.generateTreesRecu(i + 1, high)
            for j in left:
                for k in right:
                    cur = TreeNode(i)
                    cur.left = j
                    cur.right = k
                    result.append(cur)
        return result
```

remove-nth-node-from-end-of-list.py

```python
# Given a linked list, remove the n-th node from the end of list and return its
# head.
#
# Example:
#
# Given linked list: 1->2->3->4->5, and n = 2.
#
# After removing the second node from the end, the linked list becomes
# 1->2->3->5.
#
#
# Note:
#
# Given n will always be valid.
#
# Follow up:
#
# Could you do this in one pass?# Time:  O(n)
# Space: O(1)

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

    def __repr__(self):
        if self is None:
            return "Nil"
        else:
            return "{} -> {}".format(self.val, repr(self.next))

class Solution(object):
    # @return a ListNode
    def removeNthFromEnd(self, head, n):
        dummy = ListNode(-1)
        dummy.next = head
        slow, fast = dummy, dummy

        for i in xrange(n):
            fast = fast.next

        while fast.next:
            slow, fast = slow.next, fast.next

        slow.next = slow.next.next

        return dummy.next
```

valid-square.py

```python
# Given the coordinates of four points in 2D space, return whether the four
# points could construct a square.
#
# The coordinate (x,y) of a point is represented by an integer array with two
# integers.
#
# Example:
#
# Input: p1 = [0,0], p2 = [1,1], p3 = [1,0], p4 = [0,1]
# Output: True
#
#
#
#
# Note:
#
#
#       All the input integers are in the range [-10000, 10000].
#       A valid square has four equal sides with positive length and four equal
# angles (90-degree angles).
#       Input points have no order.# Time:  O(1)
# Space: O(1)

class Solution(object):
    def validSquare(self, p1, p2, p3, p4):
        """
        :type p1: List[int]
        :type p2: List[int]
        :type p3: List[int]
        :type p4: List[int]
        :rtype: bool
        """
        def dist(p1, p2):
            return (p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2

        lookup = set([dist(p1, p2), dist(p1, p3),\
                      dist(p1, p4), dist(p2, p3),\
                      dist(p2, p4), dist(p3, p4)])
        return 0 not in lookup and len(lookup) == 2
```

water-and-jug-problem.py

```python
# 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)
#
# Input: x = 3, y = 5, z = 4
# Output: True
#
#
# Example 2:
#
# Input: x = 2, y = 6, z = 5
# Output: False
#
#
# Constraints:
#
#
#       0 <= x <= 10^6
#       0 <= y <= 10^6
#       0 <= z <= 10^6# Time:  O(logn),  n is the max of (x, y)
# Space: O(1)

class Solution(object):
    def canMeasureWater(self, x, y, z):
        """
        :type x: int
        :type y: int
        :type z: int
        :rtype: bool
        """
        def gcd(a, b):
            while b:
                a, b = b, a%b
            return a

        # The problem is to solve:
        # - check z <= x + y
        # - check if there is any (a, b) integers s.t. ax + by = z
        return z == 0 or ((z <= x + y) and (z % gcd(x, y) == 0))
```

delete-columns-to-make-sorted-ii.py

```python
# Example 3:
#
# Input: ["zyx","wvu","tsr"]
# Output: 3
# Explanation:
# We have to delete every column.
#
#
#
#
#
#
# Note:
#
#
#       1 <= A.length <= 100
#       1 <= A[i].length <= 100# Time:  O(n * l)
# Space: O(n)

class Solution(object):
    def minDeletionSize(self, A):
        """
        :type A: List[str]
        :rtype: int
        """
        result = 0
        unsorted = set(range(len(A)-1))
        for j in xrange(len(A[0])):
            if any(A[i][j] > A[i+1][j] for i in unsorted):
                result += 1
            else:
                unsorted -= set(i for i in unsorted if A[i][j] < A[i+1][j])
        return result


# Time:  O(n * m)
# Space: O(n)
class Solution2(object):
    def minDeletionSize(self, A):
        """
        :type A: List[str]
        :rtype: int
        """
        result = 0
        is_sorted = [False]*(len(A)-1)
        for j in xrange(len(A[0])):
            tmp = is_sorted[:]
            for i in xrange(len(A)-1):
                if A[i][j] > A[i+1][j] and tmp[i] == False:
                    result += 1
                    break
                if A[i][j] < A[i+1][j]:
                    tmp[i] = True
            else:
                is_sorted = tmp
        return result
```

evaluate-reverse-polish-notation.py

```python
# Evaluate the value of an arithmetic expression in Reverse Polish Notation.
#
# Valid operators are +, -, *, /. Each operand may be an integer or another
# expression.
#
# Note:
#
#
#       Division between two integers should truncate toward zero.
#       The given RPN expression is always valid. That means the expression
# would always evaluate to a result and there won't be any divide by zero
# operation.
#
#
# Example 1:
#
# Input: ["2", "1", "+", "3", "*"]
# Output: 9
# Explanation: ((2 + 1) * 3) = 9
#
#
# Example 2:
#
# Input: ["4", "13", "5", "/", "+"]
# Output: 6
# Explanation: (4 + (13 / 5)) = 6
#
#
# Example 3:
#
# Input: ["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"]
# Output: 22
# Explanation:
#   ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
# = ((10 * (6 / (12 * -11))) + 17) + 5
# = ((10 * (6 / -132)) + 17) + 5
# = ((10 * 0) + 17) + 5
# = (0 + 17) + 5
# = 17 + 5
# = 22# Time:  O(n)
# Space: O(n)

import operator

class Solution(object):
    # @param tokens, a list of string
    # @return an integer
    def evalRPN(self, tokens):
        numerals, operators = [], {"+": operator.add, "-": operator.sub, "*": operator.mul, "/": operator.div}
        for token in tokens:
            if token not in operators:
                numerals.append(int(token))
            else:
                y, x = numerals.pop(), numerals.pop()
                numerals.append(int(operators[token](x * 1.0, y)))
        return numerals.pop()
```

validate-ip-address.py

```
# Given a string IP. We need to check If IP is a valid IPv4 address, valid IPv6
# address or not a valid IP address.
#
# Return "IPv4" if IP is a valid IPv4 address, "IPv6" if IP is a valid IPv6
# address or "Neither" if IP is not a valid IP of any type.
#
# A valid IPv4 address is an IP in the form "x1.x2.x3.x4" where 0 <= xi <= 255
# and xi cannot contain leading zeros. For example, "192.168.1.1"
# and "192.168.1.0" are valid IPv4 addresses but "192.168.01.1",
# "192.168.1.00" and "192.168@1.1" are invalid IPv4 adresses.
#
# A valid IPv6 address is an IP in the form "x1:x2:x3:x4:x5:x6:x7:x8" where:
#
#
#       1 <= xi.length <= 4
#       xi is hexadecimal string whcih may contain digits, lower-case English
# letter ('a' to 'f') and/or upper-case English letters ('A' to 'F').
#       Leading zeros are allowed in xi.
#
#
# For example, "2001:0db8:85a3:0000:0000:8a2e:0370:7334" and
# "2001:db8:85a3:0:0:8A2E:0370:7334" are valid IPv6 addresses but
# "2001:0db8:85a3::8A2E:037j:7334" and "02001:0db8:85a3:0000:0000:8a2e:0370:7334"
# are invalid IPv6 addresses.
#
#
# Example 1:
#
# Input: IP = "172.16.254.1"
# Output: "IPv4"
# Explanation: This is a valid IPv4 address, return "IPv4".
#
#
# Example 2:
#
# Input: IP = "2001:0db8:85a3:0:0:8A2E:0370:7334"
# Output: "IPv6"
# Explanation: This is a valid IPv6 address, return "IPv6".
#
#
# Example 3:
#
# Input: IP = "256.256.256.256"
# Output: "Neither"
# Explanation: This is neither a IPv4 address nor a IPv6 address.
#
#
# Example 4:
#
# Input: IP = "2001:0db8:85a3:0:0:8A2E:0370:7334:"
# Output: "Neither"
#
#
# Example 5:
#
# Input: IP = "1e1.4.5.6"
# Output: "Neither"
#
```

```python
#
#
# Constraints:
#
#
#       IP consists only of English letters, digits and the characters '.' and
# ':'.# Time:  O(1)
# Space: O(1)

import string


class Solution(object):
    def validIPAddress(self, IP):
        """
        :type IP: str
        :rtype: str
        """
        blocks = IP.split('.')
        if len(blocks) == 4:
            for i in xrange(len(blocks)):
                if not blocks[i].isdigit() or not 0 <= int(blocks[i]) < 256 or \
                   (blocks[i][0] == '0' and len(blocks[i]) > 1):
                    return "Neither"
            return "IPv4"

        blocks = IP.split(':')
        if len(blocks) == 8:
            for i in xrange(len(blocks)):
                if not (1 <= len(blocks[i]) <= 4) or \
                   not all(c in string.hexdigits for c in blocks[i]):
                    return "Neither"
            return "IPv6"
        return "Neither"
```

single-number-iii.py

```
# Given an integer array 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. You can return the answer in any order.
#
# Follow up: Your algorithm should run in linear runtime complexity. Could you
# implement it using only constant space complexity?
#
#
# Example 1:
#
# Input: nums = [1,2,1,3,2,5]
# Output: [3,5]
# Explanation:  [5, 3] is also a valid answer.
#
#
# Example 2:
#
# Input: nums = [-1,0]
# Output: [-1,0]
#
#
# Example 3:
#
# Input: nums = [0,1]
# Output: [1,0]
#
#
#
# Constraints:
#
#
#       1 <= nums.length <= 30000
#        Each integer in nums will appear twice, only two integers will appear
# once.# Time:  O(n)
# Space: O(1)

import operator
import collections


class Solution(object):
    # @param {integer[]} nums
    # @return {integer[]}
    def singleNumber(self, nums):
        x_xor_y = reduce(operator.xor, nums)
        bit =  x_xor_y & -x_xor_y
        result = [0, 0]
        for i in nums:
            result[bool(i & bit)] ^= i
        return result


class Solution2(object):
    # @param {integer[]} nums
    # @return {integer[]}
    def singleNumber(self, nums):
        x_xor_y = 0
        for i in nums:
```

```
            x_xor_y ^= i

        bit = x_xor_y & ~(x_xor_y - 1)

        x = 0
        for i in nums:
            if i & bit:
                x ^= i

        return [x, x ^ x_xor_y]


class Solution3(object):
    def singleNumber(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        return [x[0] for x in sorted(collections.Counter(nums).items(), key=lambda i: i[1], reverse=False)[:2]]
```

four-divisors.py

```
# Given an integer array nums, return the sum of divisors of the integers in
# that array that have exactly four divisors.
#
# If there is no such integer in the array, return 0.
#
#
# Example 1:
#
# Input: nums = [21,4,7]
# Output: 32
# Explanation:
# 21 has 4 divisors: 1, 3, 7, 21
# 4 has 3 divisors: 1, 2, 4
# 7 has 2 divisors: 1, 7
# The answer is the sum of divisors of 21 only.
#
#
#
# Constraints:
#
#
#       1 <= nums.length <= 10^4
#       1 <= nums[i] <= 10^5# Time:  O(n * sqrt(n))
# Space: O(1)

class Solution(object):
    def sumFourDivisors(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        result = 0
        for num in nums:
            facs, i = [], 1
            while i*i <= num:
                if num % i:
                    i+= 1
                    continue
                facs.append(i)
                if i != num//i:
                    facs.append(num//i)
                    if len(facs) > 4:
                        break
                i += 1
            if len(facs) == 4:
                result += sum(facs)
        return result


# Time:  O(n * sqrt(n))
# Space: O(sqrt(n))
import itertools


class Solution2(object):
    def sumFourDivisors(self, nums):
        """
        :type nums: List[int]
```

```
    :rtype: int
    """
def factorize(x):
    result = []
    d = 2
    while d*d <= x:
        e = 0
        while x%d == 0:
            x //= d
            e += 1
        if e:
            result.append([d, e])
        d += 1 if d == 2 else 2
    if x > 1:
        result.append([x, 1])
    return result

result = 0
for facs in itertools.imap(factorize, nums):
    if len(facs) == 1 and facs[0][1] == 3:
        p = facs[0][0]
        result += (p**4-1)//(p-1)  # p^0 + p^1 +p^2 +p^3
    elif len(facs) == 2 and facs[0][1] == facs[1][1] == 1:
        p, q = facs[0][0], facs[1][0]
        result += (1 + p) * (1 + q)
return result
```

next-permutation.py

```python
# 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 and use only constant 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# Time:  O(n)
# Space: O(1)

class Solution(object):
    def nextPermutation(self, nums):
        """
        :type nums: List[int]
        :rtype: None Do not return anything, modify nums in-place instead.
        """
        k, l = -1, 0
        for i in reversed(xrange(len(nums)-1)):
            if nums[i] < nums[i+1]:
                k = i
                break
        else:
            nums.reverse()
            return

        for i in reversed(xrange(k+1, len(nums))):
            if nums[i] > nums[k]:
                l = i
                break
        nums[k], nums[l] = nums[l], nums[k]
        nums[k+1:] = nums[:k:-1]


# Time:  O(n)
# Space: O(1)
class Solution2(object):
    def nextPermutation(self, nums):
        """
        :type nums: List[int]
        :rtype: None Do not return anything, modify nums in-place instead.
        """
        k, l = -1, 0
        for i in xrange(len(nums)-1):
            if nums[i] < nums[i+1]:
                k = i

        if k == -1:
            nums.reverse()
            return
```

```
for i in xrange(k+1, len(nums)):
    if nums[i] > nums[k]:
        l = i
nums[k], nums[l] = nums[l], nums[k]
nums[k+1:] = nums[:k:-1]
```

longest-string-chain.py

```
# Given a list of words, each word consists of English lowercase letters.
#
# Let's say word1 is a predecessor of word2 if and only if we can add exactly
# one letter anywhere in word1 to make it equal to word2.  For example, "abc" is a
# predecessor of "abac".
#
# A word chain is a sequence of words [word_1, word_2, ..., word_k] with k >=
# 1, where word_1 is a predecessor of word_2, word_2 is a predecessor of word_3,
# and so on.
#
# Return the longest possible length of a word chain with words chosen from the
# given list of words.
#
#
#
# Example 1:
#
# Input: ["a","b","ba","bca","bda","bdca"]
# Output: 4
# Explanation: one of the longest word chain is "a","ba","bda","bdca".
#
#
#
#
# Note:
#
#
#       1 <= words.length <= 1000
#       1 <= words[i].length <= 16
#       words[i] only consists of English lowercase letters.# Time:  O(n * l^2)
# Space: O(n * l)

import collections


class Solution(object):
    def longestStrChain(self, words):
        """
        :type words: List[str]
        :rtype: int
        """
        words.sort(key=len)
        dp = collections.defaultdict(int)
        for w in words:
            for i in xrange(len(w)):
                dp[w] = max(dp[w], dp[w[:i]+w[i+1:]]+1)
        return max(dp.itervalues())
```

reduce-array-size-to-the-half.py

```
# Given an array arr.  You can choose a set of integers and remove all the
# occurrences of these integers in the array.
#
# Return the minimum size of the set so that at least half of the integers of
# the array are removed.
#
#
# Example 1:
#
# Input: arr = [3,3,3,3,5,5,5,2,2,7]
# Output: 2
# Explanation: Choosing {3,7} will make the new array [5,5,5,2,2] which has size
# 5 (i.e equal to half of the size of the old array).
# Possible sets of size 2 are {3,5},{3,2},{5,2}.
# Choosing set {2,7} is not possible as it will make the new array
# [3,3,3,3,5,5,5] which has size greater than half of the size of the old array.
#
#
# Example 2:
#
# Input: arr = [7,7,7,7,7,7]
# Output: 1
# Explanation: The only possible set you can choose is {7}. This will make the
# new array empty.
#
#
# Example 3:
#
# Input: arr = [1,9]
# Output: 1
#
#
# Example 4:
#
# Input: arr = [1000,1000,3,7]
# Output: 1
#
#
# Example 5:
#
# Input: arr = [1,2,3,4,5,6,7,8,9,10]
# Output: 5
#
#
#
# Constraints:
#
#
#       1 <= arr.length <= 10^5
#       arr.length is even.
#       1 <= arr[i] <= 10^5# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def minSetSize(self, arr):
```

```
"""
:type arr: List[int]
:rtype: int
"""
counting_sort = [0]*len(arr)
count = collections.Counter(arr)
for c in count.itervalues():
    counting_sort[c-1] += 1
result, total = 0, 0
for c in reversed(xrange(len(arr))):
    if not counting_sort[c]:
        continue
    count = min(counting_sort[c],
                ((len(arr)+1)//2 - total - 1)//(c+1) + 1)
    result += count
    total += count*(c+1)
    if total >= (len(arr)+1)//2:
        break
return result
```

ugly-number-ii.py

```python
# Write a program to find the n-th ugly number.
#
# Ugly numbers are positive numbers whose prime factors only include 2, 3, 5.
#
# Example:
#
# Input: n = 10
# Output: 12
# Explanation: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12 is the sequence of the first 10
# ugly numbers.
#
# Note:
#
#
#       1 is typically treated as an ugly number.
#       n does not exceed 1690.# Time:  O(n)
# Space: O(1)

import heapq

class Solution(object):
    # @param {integer} n
    # @return {integer}
    def nthUglyNumber(self, n):
        ugly_number = 0

        heap = []
        heapq.heappush(heap, 1)
        for _ in xrange(n):
            ugly_number = heapq.heappop(heap)
            if ugly_number % 2 == 0:
                heapq.heappush(heap, ugly_number * 2)
            elif ugly_number % 3 == 0:
                heapq.heappush(heap, ugly_number * 2)
                heapq.heappush(heap, ugly_number * 3)
            else:
                heapq.heappush(heap, ugly_number * 2)
                heapq.heappush(heap, ugly_number * 3)
                heapq.heappush(heap, ugly_number * 5)

        return ugly_number

    def nthUglyNumber2(self, n):
        ugly = [1]
        i2 = i3 = i5 = 0
        while len(ugly) < n:
            while ugly[i2] * 2 <= ugly[-1]: i2 += 1
            while ugly[i3] * 3 <= ugly[-1]: i3 += 1
            while ugly[i5] * 5 <= ugly[-1]: i5 += 1
            ugly.append(min(ugly[i2] * 2, ugly[i3] * 3, ugly[i5] * 5))
        return ugly[-1]

    def nthUglyNumber3(self, n):
        q2, q3, q5 = [2], [3], [5]
        ugly = 1
        for u in heapq.merge(q2, q3, q5):
            if n == 1:
                return ugly
```

```python
            if u > ugly:
                ugly = u
                n -= 1
                q2 += 2 * u,
                q3 += 3 * u,
                q5 += 5 * u,


class Solution2(object):
    ugly = sorted(2**a * 3**b * 5**c
                  for a in range(32) for b in range(20) for c in range(14))

    def nthUglyNumber(self, n):
        return self.ugly[n-1]
```

design-circular-deque.py

```python
# Design your implementation of the circular double-ended queue (deque).
#
# Your implementation should support following operations:
#
#
#       MyCircularDeque(k): Constructor, set the size of the deque to be k.
#       insertFront(): Adds an item at the front of Deque. Return true if the
# operation is successful.
#       insertLast(): Adds an item at the rear of Deque. Return true if the
# operation is successful.
#       deleteFront(): Deletes an item from the front of Deque. Return true if
# the operation is successful.
#       deleteLast(): Deletes an item from the rear of Deque. Return true if the
# operation is successful.
#       getFront(): Gets the front item from the Deque. If the deque is empty,
# return -1.
#       getRear(): Gets the last item from Deque. If the deque is empty, return
# -1.
#       isEmpty(): Checks whether Deque is empty or not.
#       isFull(): Checks whether Deque is full or not.
#
#
#
#
# Example:
#
# MyCircularDeque circularDeque = new MycircularDeque(3); // set the size to be
# 3
# circularDeque.insertLast(1);                  // return true
# circularDeque.insertLast(2);                  // return true
# circularDeque.insertFront(3);                 // return true
# circularDeque.insertFront(4);                 // return false, the queue is
# full
# circularDeque.getRear();                      // return 2
# circularDeque.isFull();                               // return true
# circularDeque.deleteLast();                   // return true
# circularDeque.insertFront(4);                 // return true
# circularDeque.getFront();                     // return 4
#
#
#
#
# Note:
#
#
#       All values will be in the range of [0, 1000].
#       The number of operations will be in the range of [1, 1000].
#       Please do not use the built-in Deque library.# Time:  O(1)
# Space: O(k)

class MyCircularDeque(object):

    def __init__(self, k):
        """
        Initialize your data structure here. Set the size of the deque to be k.
        :type k: int
        """
        self.__start = 0
```

```
        self.__size = 0
        self.__buffer = [0] * k

    def insertFront(self, value):
        """
        Adds an item at the front of Deque. Return true if the operation is successful.
        :type value: int
        :rtype: bool
        """
        if self.isFull():
            return False
        self.__start = (self.__start-1) % len(self.__buffer)
        self.__buffer[self.__start] = value
        self.__size += 1
        return True

    def insertLast(self, value):
        """
        Adds an item at the rear of Deque. Return true if the operation is successful.
        :type value: int
        :rtype: bool
        """
        if self.isFull():
            return False
        self.__buffer[(self.__start+self.__size) % len(self.__buffer)] = value
        self.__size += 1
        return True

    def deleteFront(self):
        """
        Deletes an item from the front of Deque. Return true if the operation is successful.
        :rtype: bool
        """
        if self.isEmpty():
            return False
        self.__start = (self.__start+1) % len(self.__buffer)
        self.__size -= 1
        return True

    def deleteLast(self):
        """
        Deletes an item from the rear of Deque. Return true if the operation is successful.
        :rtype: bool
        """
        if self.isEmpty():
            return False
        self.__size -= 1
        return True

    def getFront(self):
        """
        Get the front item from the deque.
        :rtype: int
        """
        return -1 if self.isEmpty() else self.__buffer[self.__start]

    def getRear(self):
        """
        Get the last item from the deque.
        :rtype: int
```

```
        """
        return -1 if self.isEmpty() else self.__buffer[(self.__start+self.__size-1) % len(self.__buffer)]

    def isEmpty(self):
        """
        Checks whether the circular deque is empty or not.
        :rtype: bool
        """
        return self.__size == 0

    def isFull(self):
        """
        Checks whether the circular deque is full or not.
        :rtype: bool
        """
        return self.__size == len(self.__buffer)
```

smallest-integer-divisible-by-k.py

```python
# Given a positive integer K, you need find the smallest positive integer N such
# that N is divisible by K, and N only contains the digit 1.
#
# Return the length of N.  If there is no such N, return -1.
#
#
#
# Example 1:
#
# Input: 1
# Output: 1
# Explanation: The smallest answer is N = 1, which has length 1.
#
# Example 2:
#
# Input: 2
# Output: -1
# Explanation: There is no such positive integer N divisible by 2.
#
# Example 3:
#
# Input: 3
# Output: 3
# Explanation: The smallest answer is N = 111, which has length 3.
#
#
#
# Note:
#
#
#       1 <= K <= 10^5# Time:  O(k)
# Space: O(1)

class Solution(object):
    def smallestRepunitDivByK(self, K):
        """
        :type K: int
        :rtype: int
        """
        # by observation, K % 2 = 0 or K % 5 = 0, it is impossible
        if K % 2 == 0 or K % 5 == 0:
            return -1

        # let f(N) is a N-length integer only containing digit 1
        # if there is no N in range (1..K) s.t. f(N) % K = 0
        # => there must be K remainders of f(N) % K in range (1..K-1) excluding 0
        # => due to pigeonhole principle, there must be at least 2 same remainders
        # => there must be some x, y in range (1..K) and x > y s.t. f(x) % K = f(y) % K
        # => (f(x) - f(y)) % K = 0
        # => (f(x-y) * 10^y) % K = 0
        # => due to (x-y) in range (1..K)
        # => f(x-y) % K != 0
        # => 10^y % K = 0
        # => K % 2 = 0 or K % 5 = 0
        # => -><-
        # it proves that there must be some N in range (1..K) s.t. f(N) % K = 0
        result = 0
        for N in xrange(1, K+1):
```

```python
        result = (result*10+1) % K
        if not result:
            return N
    assert(False)
    return -1  # never reach
```

dota2-senate.py

```
# In the world of Dota2, there are two parties: the Radiant and the Dire.
#
# The Dota2 senate consists of senators coming from two parties. Now the senate
# wants to make a decision about a change in the Dota2 game. The voting for this
# change is a round-based procedure. In each round, each senator can exercise one
# of the two rights:
#
#
#       Ban one senator's right:
#
#       A senator can make another senator lose all his rights in this and all
# the following rounds.
#       Announce the victory:
#
#       If this senator found the senators who still have rights to vote are all
# from the same party, he can announce the victory and make the decision about the
# change in the game.
#
#
#
#
# Given a string representing each senator's party belonging. The character 'R'
# and 'D' represent the Radiant party and the Dire party respectively. Then if
# there are n senators, the size of the given string will be n.
#
# The round-based procedure starts from the first senator to the last senator in
# the given order. This procedure will last until the end of voting. All the
# senators who have lost their rights will be skipped during the procedure.
#
# Suppose every senator is smart enough and will play the best strategy for his
# own party, you need to predict which party will finally announce the victory and
# make the change in the Dota2 game. The output should be Radiant or Dire.
#
# Example 1:
#
# Input: "RD"
# Output: "Radiant"
# Explanation: The first senator comes from Radiant and he can just ban the next
# senator's right in the round 1.
# And the second senator can't exercise any rights any more since his right has
# been banned.
# And in the round 2, the first senator can just announce the victory since he
# is the only guy in the senate who can vote.
#
#
#
#
# Example 2:
#
# Input: "RDD"
# Output: "Dire"
# Explanation:
# The first senator comes from Radiant and he can just ban the next senator's
# right in the round 1.
# And the second senator can't exercise any rights anymore since his right has
# been banned.
# And the third senator comes from Dire and he can ban the first senator's right
# in the round 1.
```

```
# And in the round 2, the third senator can just announce the victory since he
# is the only guy in the senate who can vote.
#
#
#
#
# Note:
#
#
#       The length of the given string will in the range [1, 10,000].# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def predictPartyVictory(self, senate):
        """
        :type senate: str
        :rtype: str
        """
        n = len(senate)
        radiant, dire = collections.deque(), collections.deque()
        for i, c in enumerate(senate):
            if c == 'R':
                radiant.append(i)
            else:
                dire.append(i)
        while radiant and dire:
            r_idx, d_idx = radiant.popleft(), dire.popleft()
            if r_idx < d_idx:
                radiant.append(r_idx+n)
            else:
                dire.append(d_idx+n)
        return "Radiant" if len(radiant) > len(dire) else "Dire"
```

wiggle-subsequence.py

```
# A sequence of numbers is called a wiggle sequence if the differences between
# successive numbers strictly alternate between positive and negative. The first
# difference (if one exists) may be either positive or negative. A sequence with
# fewer than two elements is trivially a wiggle sequence.
#
# For example, [1,7,4,9,2,5] is a wiggle sequence because the differences
# (6,-3,5,-7,3) are alternately positive and negative. In contrast, [1,4,7,2,5]
# and [1,7,4,5,5] are not wiggle sequences, the first because its first two
# differences are positive and the second because its last difference is zero.
#
# Given a sequence of integers, return the length of the longest subsequence
# that is a wiggle sequence. A subsequence is obtained by deleting some number of
# elements (eventually, also zero) from the original sequence, leaving the
# remaining elements in their original order.
#
# Example 1:
#
# Input: [1,7,4,9,2,5]
# Output: 6
# Explanation: The entire sequence is a wiggle sequence.
#
#
# Example 2:
#
# Input: [1,17,5,10,13,15,10,5,16,8]
# Output: 7
# Explanation: There are several subsequences that achieve this length. One is
# [1,17,10,13,10,16,8].
#
#
# Example 3:
#
# Input: [1,2,3,4,5,6,7,8,9]
# Output: 2
#
# Follow up:
#
# Can you do it in O(n) time?# Time:  O(n)
# Space: O(1)

class Solution(object):
    def wiggleMaxLength(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if len(nums) < 2:
            return len(nums)

        length, up = 1, None

        for i in xrange(1, len(nums)):
            if nums[i - 1] < nums[i] and (up is None or up is False):
                length += 1
                up = True
            elif nums[i - 1] > nums[i] and (up is None or up is True):
                length += 1
                up = False
```

```
return length
```

gas-station.py

```python
# 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 in the clockwise direction, otherwise return -1.
#
# Note:
#
#
#       If there exists a solution, it is guaranteed to be unique.
#       Both input arrays are non-empty and have the same length.
#       Each element in the input arrays is a non-negative integer.
#
#
# Example 1:
#
# Input:
# gas  = [1,2,3,4,5]
# cost = [3,4,5,1,2]
#
# Output: 3
#
# Explanation:
# Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4
# = 4
# Travel to station 4. Your tank = 4 - 1 + 5 = 8
# Travel to station 0. Your tank = 8 - 2 + 1 = 7
# Travel to station 1. Your tank = 7 - 3 + 2 = 6
# Travel to station 2. Your tank = 6 - 4 + 3 = 5
# Travel to station 3. The cost is 5. Your gas is just enough to travel back to
# station 3.
# Therefore, return 3 as the starting index.
#
#
# Example 2:
#
# Input:
# gas  = [2,3,4]
# cost = [3,4,3]
#
# Output: -1
#
# Explanation:
# You can't start at station 0 or 1, as there is not enough gas to travel to the
# next station.
# Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
# Travel to station 0. Your tank = 4 - 3 + 2 = 3
# Travel to station 1. Your tank = 3 - 3 + 3 = 3
# You cannot travel back to station 2, as it requires 4 unit of gas but you only
# have 3.
# Therefore, you can't travel around the circuit once no matter where you start.# Time:  O(n)
# Space: O(1)

class Solution(object):
```

```python
# @param gas, a list of integers
# @param cost, a list of integers
# @return an integer
def canCompleteCircuit(self, gas, cost):
    start, total_sum, current_sum = 0, 0, 0
    for i in xrange(len(gas)):
        diff = gas[i] - cost[i]
        current_sum += diff
        total_sum += diff
        if current_sum < 0:
            start = i + 1
            current_sum = 0
    if total_sum >= 0:
        return start

    return -1
```

deepest-leaves-sum.py

```python
# Given a binary tree, return the sum of values of its deepest leaves.
#
# Example 1:
#
#
#
# Input: root = [1,2,3,4,5,null,6,7,null,null,null,null,8]
# Output: 15
#
#
#
# Constraints:
#
#
#       The number of nodes in the tree is between 1 and 10^4.
#       The value of nodes is between 1 and 100.# Time:  O(n)
# Space: O(w)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def deepestLeavesSum(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        curr = [root]
        while curr:
            prev, curr = curr, [child for p in curr for child in [p.left, p.right] if child]
        return sum(node.val for node in prev)
```

check-if-word-is-valid-after-substitutions.py

```python
# We can say that a string is valid if it follows one of the three following
# cases:
#
#
#       An empty string "" is valid.
#       The string "abc" is also valid.
#       Any string in the form "a" + str + "bc", "ab" + str + "c", str + "abc"
# or "abc" + str where str is a valid string is also considered a valid string.
#
#
# For example, "", "abc", "aabcbc", "abcabc" and "abcabcababcc" are all valid
# strings, while "abccba", "ab", "cababc" and "bac" are not valid strings.
#
# Given a string s, return true if it is a valid string, otherwise, return
# false.
#
#
# Example 1:
#
# Input: s = "aabcbc"
# Output: true
# Explanation:
# We start with the valid string "abc".
# Then we can insert another "abc" between "a" and "bc", resulting in "a" +
# "abc" + "bc" which is "aabcbc".
#
#
# Example 2:
#
# Input: s = "abcabcababcc"
# Output: true
# Explanation:
# "abcabcabc" is valid after consecutive insertings of "abc".
# Then we can insert "abc" before the last letter, resulting in "abcabcab" +
# "abc" + "c" which is "abcabcababcc".
#
#
# Example 3:
#
# Input: s = "abccba"
# Output: false
#
#
# Example 4:
#
# Input: s = "cababc"
# Output: false
#
#
#
# Constraints:
#
#
#       1 <= s.length <= 2 * 104
#       s[i] is 'a', 'b', or 'c'# Time:  O(n)
# Space: O(n)

class Solution(object):
```

```python
def isValid(self, S):
    """
    :type S: str
    :rtype: bool
    """
    stack = []
    for i in S:
        if i == 'c':
            if stack[-2:] == ['a', 'b']:
                stack.pop()
                stack.pop()
            else:
                return False
        else:
            stack.append(i)
    return not stack
```

subsets-ii.py

```python
# Given a collection of integers that might contain duplicates, nums, return all
# possible subsets (the power set).
#
# Note: The solution set must not contain duplicate subsets.
#
# Example:
#
# Input: [1,2,2]
# Output:
# [
#   [2],
#   [1],
#   [1,2,2],
#   [2,2],
#   [1,2],
#   []
# ]# Time:  O(n * 2^n)
# Space: O(1)

class Solution(object):
    def subsetsWithDup(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        nums.sort()
        result = [[]]
        previous_size = 0
        for i in xrange(len(nums)):
            size = len(result)
            for j in xrange(size):
                # Only union non-duplicate element or new union set.
                if i == 0 or nums[i] != nums[i - 1] or j >= previous_size:
                    result.append(list(result[j]))
                    result[-1].append(nums[i])
            previous_size = size
        return result


# Time:  O(n * 2^n) ~ O((n * 2^n)^2)
# Space: O(1)
class Solution2(object):
    def subsetsWithDup(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        result = []
        i, count = 0, 1 << len(nums)
        nums.sort()

        while i < count:
            cur = []
            for j in xrange(len(nums)):
                if i & 1 << j:
                    cur.append(nums[j])
            if cur not in result:
                result.append(cur)
```

```python
            i += 1

        return result


# Time:  O(n * 2^n) ~ O((n * 2^n)^2)
# Space: O(1)
class Solution3(object):
    def subsetsWithDup(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        result = []
        self.subsetsWithDupRecu(result, [], sorted(nums))
        return result

    def subsetsWithDupRecu(self, result, cur, nums):
        if not nums:
            if cur not in result:
                result.append(cur)
        else:
            self.subsetsWithDupRecu(result, cur, nums[1:])
            self.subsetsWithDupRecu(result, cur + [nums[0]], nums[1:])
```

insufficient-nodes-in-root-to-leaf-paths.py

```python
# Example 2:
#
#
# Input: root = [5,4,8,11,null,17,4,7,1,null,null,5,3], limit = 22
#
# Output: [5,4,8,11,null,17,4,7,null,null,null,5]
#
#
#
# Example 3:
#
#
# Input: root = [1,2,-3,-5,null,4,null], limit = -1
#
# Output: [1,null,-3,4]# Time:  O(n)
# Space: O(h)

# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def sufficientSubset(self, root, limit):
        """
        :type root: TreeNode
        :type limit: int
        :rtype: TreeNode
        """
        if not root:
            return None
        if not root.left and not root.right:
            return None if root.val < limit else root
        root.left = self.sufficientSubset(root.left, limit-root.val)
        root.right = self.sufficientSubset(root.right, limit-root.val)
        if not root.left and not root.right:
            return None
        return root
```

maximum-binary-tree-ii.py

```
# We are given the root node of a maximum tree: a tree where every node has a
# value greater than any other value in its subtree.
#
# Just as in the previous problem, the given tree was constructed from an
# list A (root = Construct(A)) recursively with the following Construct(A)
# routine:
#
#
#       If A is empty, return null.
#       Otherwise, let A[i] be the largest element of A.  Create a root node
# with value A[i].
#       The left child of root will be Construct([A[0], A[1], ..., A[i-1]])
#       The right child of root will be Construct([A[i+1], A[i+2], ...,
# A[A.length - 1]])
#       Return root.
#
#
# Note that we were not given A directly, only a root node root = Construct(A).
#
# Suppose B is a copy of A with the value val appended to it.  It is guaranteed
# that B has unique values.
#
# Return Construct(B).
#
#
# Example 1:
#
#
#
# Input: root = [4,1,3,null,null,2], val = 5
# Output: [5,4,null,1,3,null,null,2]
# Explanation: A = [1,4,2,3], B = [1,4,2,3,5]
#
#
# Example 2:
#
#
#
# Input: root = [5,2,4,null,1], val = 3
# Output: [5,2,4,null,1,null,3]
# Explanation: A = [2,1,5,4], B = [2,1,5,4,3]
#
#
# Example 3:
#
#
#
# Input: root = [5,2,3,null,1], val = 4
# Output: [5,2,4,null,1,3]
# Explanation: A = [2,1,5,3], B = [2,1,5,3,4]
#
#
#
# Constraints:
#
#
#       1 <= B.length <= 100# Time:  O(h)
# Space: O(1)
```

```python
# Definition for a binary tree node.
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def insertIntoMaxTree(self, root, val):
        """
        :type root: TreeNode
        :type val: int
        :rtype: TreeNode
        """
        if not root:
            return TreeNode(val)

        if val > root.val:
            node = TreeNode(val)
            node.left = root
            return node

        curr = root
        while curr.right and curr.right.val > val:
            curr = curr.right
        node = TreeNode(val)
        curr.right, node.left = node, curr.right
        return root
```

longest-substring-with-at-least-k-repeating-characters.py

```python
# Find the length of the longest substring T of a given string (consists of
# lowercase letters only) such that every character in T appears no less than k
# times.
#
#
# Example 1:
# Input:
# s = "aaabb", k = 3
#
# Output:
# 3
#
# The longest substring is "aaa", as 'a' is repeated 3 times.
#
#
#
# Example 2:
# Input:
# s = "ababbc", k = 2
#
# Output:
# 5
#
# The longest substring is "ababb", as 'a' is repeated 2 times and 'b' is
# repeated 3 times.# Time:  O(26 * n) = O(n)
# Space: O(26) = O(1)

class Solution(object):
    def longestSubstring(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: int
        """
        def longestSubstringHelper(s, k, start, end):
            count = [0] * 26
            for i in xrange(start, end):
                count[ord(s[i]) - ord('a')] += 1
            max_len = 0
            i = start
            while i < end:
                while i < end and count[ord(s[i]) - ord('a')] < k:
                    i += 1
                j = i
                while j < end and count[ord(s[j]) - ord('a')] >= k:
                    j += 1

                if i == start and j == end:
                    return end - start

                max_len = max(max_len, longestSubstringHelper(s, k, i, j))
                i = j
            return max_len

        return longestSubstringHelper(s, k, 0, len(s))
```

diagonal-traverse.py

```python
# Given a matrix of M x N elements (M rows, N columns), return all elements of
# the matrix in diagonal order as shown in the below image.
#
#
#
# Example:
#
# Input:
# [
#  [ 1, 2, 3 ],
#  [ 4, 5, 6 ],
#  [ 7, 8, 9 ]
# ]
#
# Output:  [1,2,4,7,5,3,6,8,9]
#
# Explanation:
#
#
#
#
#
# Note:
#
# The total number of elements of the given matrix will not exceed 10,000.# Time:  O(m * n)
# Space: O(1)

class Solution(object):
    def findDiagonalOrder(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: List[int]
        """
        if not matrix or not matrix[0]:
            return []

        result = []
        row, col, d = 0, 0, 0
        dirs = [(-1, 1), (1, -1)]

        for i in xrange(len(matrix) * len(matrix[0])):
            result.append(matrix[row][col])
            row += dirs[d][0]
            col += dirs[d][1]

            if row >= len(matrix):
                row = len(matrix) - 1
                col += 2
                d = 1 - d
            elif col >= len(matrix[0]):
                col = len(matrix[0]) - 1
                row += 2
                d = 1 - d
            elif row < 0:
                row = 0
                d = 1 - d
            elif col < 0:
                col = 0
```

```
        d = 1 - d

return result
```

evaluate-division.py

```
# Equations are given in the format A / B = k, where A and B are variables
# represented as strings, and k is a real number (floating point number). Given
# some queries, return the answers. If the answer does not exist, return -1.0.
#
# Example:
#
# Given  a / b = 2.0, b / c = 3.0.
#
# queries are:  a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ? .
#
# return  [6.0, 0.5, -1.0, 1.0, -1.0 ].
#
# The input is:  vector<pair<string, string>> equations, vector<double>& values,
# vector<pair<string, string>> queries , where equations.size() == values.size(),
# and the values are positive. This represents the equations. Return
# vector<double>.
#
# According to the example above:
#
# equations = [ ["a", "b"], ["b", "c"] ],
# values = [2.0, 3.0],
# queries = [ ["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"] ].
#
#
#
# The input is always valid. You may assume that evaluating the queries will
# result in no division by zero and there is no contradiction.# Time:  O(e + q * |V|!), |V| is the number of v
# Space: O(e)

import collections


class Solution(object):
    def calcEquation(self, equations, values, query):
        """
        :type equations: List[List[str]]
        :type values: List[float]
        :type query: List[List[str]]
        :rtype: List[float]
        """
        def check(up, down, lookup, visited):
            if up in lookup and down in lookup[up]:
                return (True, lookup[up][down])
            for k, v in lookup[up].iteritems():
                if k not in visited:
                    visited.add(k)
                    tmp = check(k, down, lookup, visited)
                    if tmp[0]:
                        return (True, v * tmp[1])
            return (False, 0)

        lookup = collections.defaultdict(dict)
        for i, e in enumerate(equations):
            lookup[e[0]][e[1]] = values[i]
            if values[i]:
                lookup[e[1]][e[0]] = 1.0 / values[i]

        result = []
```

```python
for q in query:
    visited = set()
    tmp = check(q[0], q[1], lookup, visited)
    result.append(tmp[1] if tmp[0] else -1)
return result
```

minimum-time-difference.py

```python
# Given a list of 24-hour clock time points in "Hour:Minutes" format, find the
# minimum minutes difference between any two time points in the list.
#
# Example 1:
#
# Input: ["23:59","00:00"]
# Output: 1
#
#
#
# Note:
#
#
# The number of time points in the given list is at least 2 and won't exceed
# 20000.
# The input time is legal and ranges from 00:00 to 23:59.# Time:  O(nlogn)
# Space: O(n)

class Solution(object):
    def findMinDifference(self, timePoints):
        """
        :type timePoints: List[str]
        :rtype: int
        """
        minutes = map(lambda x: int(x[:2]) * 60 + int(x[3:]), timePoints)
        minutes.sort()
        return min((y - x) % (24 * 60)  \
                   for x, y in zip(minutes, minutes[1:] + minutes[:1]))
```

binary-tree-zigzag-level-order-traversal.py

```python
# 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],
#
#     3
#    / \
#   9  20
#     /  \
#    15   7
#
#
#
# return its zigzag level order traversal as:
#
# [
#   [3],
#   [20,9],
#   [15,7]
# ]# Time:  O(n)
# Space: O(n)

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    # @param root, a tree node
    # @return a list of lists of integers
    def zigzagLevelOrder(self, root):
        if root is None:
            return []
        result, current = [], [root]
        while current:
            next_level, vals = [], []
            for node in current:
                vals.append(node.val)
                if node.left:
                    next_level.append(node.left)
                if node.right:
                    next_level.append(node.right)
            result.append(vals[::-1] if len(result) % 2 else vals)
            current = next_level
        return result
```

minimum-score-triangulation-of-polygon.py

```python
# Given N, consider a convex N-sided polygon with vertices labelled A[0], A[i],
# ..., A[N-1] in clockwise order.
#
# Suppose you triangulate the polygon into N-2 triangles.  For each triangle,
# the value of that triangle is the product of the labels of the vertices, and the
# total score of the triangulation is the sum of these values over all N-2
# triangles in the triangulation.
#
# Return the smallest possible total score that you can achieve with some
# triangulation of the polygon.
#
#
#
#
#
#
#
# Example 1:
#
# Input: [1,2,3]
# Output: 6
# Explanation: The polygon is already triangulated, and the score of the only
# triangle is 6.
#
#
#
# Example 2:
#
#
#
# Input: [3,7,4,5]
# Output: 144
# Explanation: There are two triangulations, with possible scores: 3*7*5 + 4*5*7
# = 245, or 3*4*5 + 3*4*7 = 144.  The minimum score is 144.
#
#
#
# Example 3:
#
# Input: [1,3,1,4,1,5]
# Output: 13
# Explanation: The minimum score triangulation has score 1*1*3 + 1*1*4 + 1*1*5 +
# 1*1*1 = 13.
#
#
#
#
# Note:
#
#
#       3 <= A.length <= 50
#       1 <= A[i] <= 100# Time:  O(n^3)
# Space: O(n^2)

class Solution(object):
    def minScoreTriangulation(self, A):
        """
        :type A: List[int]
```

```
    :rtype: int
    """
    dp = [[0 for _ in xrange(len(A))] for _ in xrange(len(A))]
    for p in xrange(3, len(A)+1):
        for i in xrange(len(A)-p+1):
            j = i+p-1;
            dp[i][j] = float("inf")
            for k in xrange(i+1, j):
                dp[i][j] = min(dp[i][j], dp[i][k]+dp[k][j] + A[i]*A[j]*A[k])
    return dp[0][-1]
```

multiply-strings.py

```python
# Given two non-negative integers num1 and num2 represented as strings, return
# the product of num1 and num2, also represented as a string.
#
# Example 1:
#
# Input: num1 = "2", num2 = "3"
# Output: "6"
#
# Example 2:
#
# Input: num1 = "123", num2 = "456"
# Output: "56088"
#
#
# Note:
#
#
#       The length of both num1 and num2 is < 110.
#       Both num1 and num2 contain only digits 0-9.
#       Both num1 and num2 do not contain any leading zero, except the number 0
# itself.
#       You must not use any built-in BigInteger library or convert the inputs
# to integer directly.# Time:  O(m * n)
# Space: O(m + n)

class Solution(object):
    def multiply(self, num1, num2):
        """
        :type num1: str
        :type num2: str
        :rtype: str
        """
        num1, num2 = num1[::-1], num2[::-1]
        res = [0] * (len(num1) + len(num2))
        for i in xrange(len(num1)):
            for j in xrange(len(num2)):
                res[i + j] += int(num1[i]) * int(num2[j])
                res[i + j + 1] += res[i + j] / 10
                res[i + j] %= 10

        # Skip leading 0s.
        i = len(res) - 1
        while i > 0 and res[i] == 0:
            i -= 1

        return ''.join(map(str, res[i::-1]))

# Time:  O(m * n)
# Space: O(m + n)
# Using built-in bignum solution.
class Solution2(object):
    def multiply(self, num1, num2):
        """
        :type num1: str
        :type num2: str
        :rtype: str
        """
        return str(int(num1) * int(num2))
```

circular-array-loop.py

```python
# You are given a circular array nums of positive and negative integers. If a
# number k at an index is positive, then move forward k steps. Conversely, if it's
# negative (-k), move backward k steps. Since the array is circular, you may
# assume that the last element's next element is the first element, and the first
# element's previous element is the last element.
#
# Determine if there is a loop (or a cycle) in nums. A cycle must start and end
# at the same index and the cycle's length > 1. Furthermore, movements in a cycle
# must all follow a single direction. In other words, a cycle must not consist of
# both forward and backward movements.
#
#
#
# Example 1:
#
# Input: [2,-1,1,2,2]
# Output: true
# Explanation: There is a cycle, from index 0 -> 2 -> 3 -> 0. The cycle's length
# is 3.
#
#
# Example 2:
#
# Input: [-1,2]
# Output: false
# Explanation: The movement from index 1 -> 1 -> 1 ... is not a cycle, because
# the cycle's length is 1. By definition the cycle's length must be greater than
# 1.
#
#
# Example 3:
#
# Input: [-2,1,-1,-2,-2]
# Output: false
# Explanation: The movement from index 1 -> 2 -> 1 -> ... is not a cycle,
# because movement from index 1 -> 2 is a forward movement, but movement from
# index 2 -> 1 is a backward movement. All movements in a cycle must follow a
# single direction.
#
#
#
# Note:
#
#
#       -1000  nums[i]  1000
#       nums[i]  0
#       1  nums.length  5000
#
#
#
#
# Follow up:
#
# Could you solve it in O(n) time complexity and O(1) extra space complexity?# Time:  O(n)
# Space: O(1)

class Solution(object):
    def circularArrayLoop(self, nums):
```

```python
"""
:type nums: List[int]
:rtype: bool
"""
def next_index(nums, i):
    return (i + nums[i]) % len(nums)

for i in xrange(len(nums)):
    if nums[i] == 0:
        continue

    slow, fast = i, i
    while nums[next_index(nums, slow)] * nums[i] > 0 and \
          nums[next_index(nums, fast)] * nums[i] > 0 and \
          nums[next_index(nums, next_index(nums, fast))] * nums[i] > 0:
        slow = next_index(nums, slow)
        fast = next_index(nums, next_index(nums, fast))
        if slow == fast:
            if slow == next_index(nums, slow):
                break
            return True

    slow, val = i, nums[i]
    while nums[slow] * val > 0:
        tmp = next_index(nums, slow)
        nums[slow] = 0
        slow = tmp

return False
```

largest-1-bordered-square.py

```python
# Given a 2D grid of 0s and 1s, return the number of elements in the largest
# square subgrid that has all 1s on its border, or 0 if such a subgrid doesn't
# exist in the grid.
#
#
# Example 1:
#
# Input: grid = [[1,1,1],[1,0,1],[1,1,1]]
# Output: 9
#
#
# Example 2:
#
# Input: grid = [[1,1,0,0]]
# Output: 1
#
#
#
# Constraints:
#
#
#       1 <= grid.length <= 100
#       1 <= grid[0].length <= 100
#       grid[i][j] is 0 or 1# Time:  O(n^3)
# Space: O(n^2)

class Solution(object):
    def largest1BorderedSquare(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        top, left = [a[:] for a in grid], [a[:] for a in grid]
        for i in xrange(len(grid)):
            for j in xrange(len(grid[0])):
                if not grid[i][j]:
                    continue
                if i:
                    top[i][j] = top[i-1][j] + 1
                if j:
                    left[i][j] = left[i][j-1] + 1
        for l in reversed(xrange(1, min(len(grid), len(grid[0]))+1)):
            for i in xrange(len(grid)-l+1):
                for j in xrange(len(grid[0])-l+1):
                    if min(top[i+l-1][j],
                           top[i+l-1][j+l-1],
                           left[i][j+l-1],
                           left[i+l-1][j+l-1]) >= l:
                        return l*l
        return 0
```

check-if-a-string-contains-all-binary-codes-of-size-k.py

```python
# Given a binary string s and an integer k.
#
# Return True if every binary code of length k is a substring of s. Otherwise,
# return False.
#
#
# Example 1:
#
# Input: s = "00110110", k = 2
# Output: true
# Explanation: The binary codes of length 2 are "00", "01", "10" and "11". They
# can be all found as substrings at indicies 0, 1, 3 and 2 respectively.
#
#
# Example 2:
#
# Input: s = "00110", k = 2
# Output: true
#
#
# Example 3:
#
# Input: s = "0110", k = 1
# Output: true
# Explanation: The binary codes of length 1 are "0" and "1", it is clear that
# both exist as a substring.
#
#
# Example 4:
#
# Input: s = "0110", k = 2
# Output: false
# Explanation: The binary code "00" is of length 2 and doesn't exist in the
# array.
#
#
# Example 5:
#
# Input: s = "0000000001011100", k = 4
# Output: false
#
#
#
# Constraints:
#
#
#       1 <= s.length <= 5 * 10^5
#       s consists of 0's and 1's only.
#       1 <= k <= 20# Time:  O(n * k)
# Space: O(k * 2^k)

class Solution(object):
    def hasAllCodes(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: bool
        """
```

```python
        return 2**k <= len(s) and len({s[i:i+k] for i in xrange(len(s)-k+1)}) == 2**k


# Time:  O(n * k)
# Space: O(2^k)
class Solution2(object):
    def hasAllCodes(self, s, k):
        """
        :type s: str
        :type k: int
        :rtype: bool
        """
        lookup = set()
        base = 2**k
        if base > len(s):
            return False
        num = 0
        for i in xrange(len(s)):
            num = (num << 1) + (s[i] == '1')
            if i >= k-1:
                lookup.add(num)
                num -= (s[i-k+1] == '1') * (base//2)
        return len(lookup) == base
```

valid-sudoku.py

```
# Determine if a 9x9 Sudoku board is valid. Only the filled cells need to be
# validated according to the following rules:
#
#
#       Each row must contain the digits 1-9 without repetition.
#       Each column must contain the digits 1-9 without repetition.
#       Each of the 9 3x3 sub-boxes of the grid must contain the
# digits 1-9 without repetition.
#
#
#
#
# A partially filled sudoku which is valid.
#
# The Sudoku board could be partially filled, where empty cells are filled with
# the character '.'.
#
# Example 1:
#
# Input:
# [
#   ["5","3",".",".","7",".",".",".","."],
#   ["6",".",".","1","9","5",".",".","."],
#   [".","9","8",".",".",".",".","6","."],
#   ["8",".",".",".","6",".",".",".","3"],
#   ["4",".",".","8",".","3",".",".","1"],
#   ["7",".",".",".","2",".",".",".","6"],
#   [".","6",".",".",".",".","2","8","."],
#   [".",".",".","4","1","9",".",".","5"],
#   [".",".",".",".","8",".",".","7","9"]
# ]
# Output: true
#
#
# Example 2:
#
# Input:
# [
#   ["8","3",".",".","7",".",".",".","."],
#   ["6",".",".","1","9","5",".",".","."],
#   [".","9","8",".",".",".",".","6","."],
#   ["8",".",".",".","6",".",".",".","3"],
#   ["4",".",".","8",".","3",".",".","1"],
#   ["7",".",".",".","2",".",".",".","6"],
#   [".","6",".",".",".",".","2","8","."],
#   [".",".",".","4","1","9",".",".","5"],
#   [".",".",".",".","8",".",".","7","9"]
# ]
# Output: false
# Explanation: Same as Example 1, except with the 5 in the top left corner being
#     modified to 8. Since there are two 8's in the top left 3x3 sub-box, it is
# invalid.
#
#
# Note:
#
#
#       A Sudoku board (partially filled) could be valid but is not necessarily
```

```python
# solvable.
#       Only the filled cells need to be validated according to the
# mentioned rules.
#       The given board contain only digits 1-9 and the character '.'.
#       The given board size is always 9x9.# Time:  O(9^2)
# Space: O(9)

class Solution(object):
    def isValidSudoku(self, board):
        """
        :type board: List[List[str]]
        :rtype: bool
        """
        for i in xrange(9):
            if not self.isValidList([board[i][j] for j in xrange(9)]) or \
               not self.isValidList([board[j][i] for j in xrange(9)]):
                return False
        for i in xrange(3):
            for j in xrange(3):
                if not self.isValidList([board[m][n] for n in xrange(3 * j, 3 * j + 3) \
                                                     for m in xrange(3 * i, 3 * i + 3)]):
                    return False
        return True

    def isValidList(self, xs):
        xs = filter(lambda x: x != '.', xs)
        return len(set(xs)) == len(xs)
```

can-i-win.py

```
# In the "100 game" two players take turns adding, to a running total, any
# integer from 1 to 10. The player who first causes the running total to reach or
# exceed 100 wins.
#
# What if we change the game so that players cannot re-use integers?
#
# For example, two players might take turns drawing from a common pool of
# numbers from 1 to 15 without replacement until they reach a total >= 100.
#
# Given two integers maxChoosableInteger and desiredTotal, return true if the
# first player to move can force a win, otherwise return false. Assume both
# players play optimally.
#
#
# Example 1:
#
# Input: maxChoosableInteger = 10, desiredTotal = 11
# Output: false
# Explanation:
# No matter which integer the first player choose, the first player will lose.
# The first player can choose an integer from 1 up to 10.
# If the first player choose 1, the second player can only choose integers from
# 2 up to 10.
# The second player will win by choosing 10 and get a total = 11, which is >=
# desiredTotal.
# Same with other integers chosen by the first player, the second player will
# always win.
#
#
# Example 2:
#
# Input: maxChoosableInteger = 10, desiredTotal = 0
# Output: true
#
#
# Example 3:
#
# Input: maxChoosableInteger = 10, desiredTotal = 1
# Output: true
#
#
#
# Constraints:
#
#
#       1 <= maxChoosableInteger <= 20
#       0 <= desiredTotal <= 300# Time:  O(n!)
# Space: O(n)

class Solution(object):
    def canIWin(self, maxChoosableInteger, desiredTotal):
        """
        :type maxChoosableInteger: int
        :type desiredTotal: int
        :rtype: bool
        """
        def canIWinHelper(maxChoosableInteger, desiredTotal, visited, lookup):
            if visited in lookup:
```

```
            return lookup[visited]

        mask = 1
        for i in xrange(maxChoosableInteger):
            if visited & mask == 0:
                if i + 1 >= desiredTotal or \
                   not canIWinHelper(maxChoosableInteger, desiredTotal - (i + 1), visited | mask, lookup):
                    lookup[visited] = True
                    return True
            mask <<= 1
        lookup[visited] = False
        return False

    if (1 + maxChoosableInteger) * (maxChoosableInteger / 2) < desiredTotal:
        return False

    return canIWinHelper(maxChoosableInteger, desiredTotal, 0, {})
```

additive-number.py

```python
# Additive number is a string whose digits can form additive sequence.
#
# A valid additive sequence should contain at least three numbers. Except for
# the first two numbers, each subsequent number in the sequence must be the sum of
# the preceding two.
#
# Given a string containing only digits '0'-'9', write a function to determine
# if it's an additive number.
#
# Note: Numbers in the additive sequence cannot have leading zeros, so sequence
# 1, 2, 03 or 1, 02, 3 is invalid.
#
#
# Example 1:
#
# Input: "112358"
# Output: true
# Explanation: The digits can form an additive sequence: 1, 1, 2, 3, 5, 8.
#              1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8
#
#
# Example 2:
#
# Input: "199100199"
# Output: true
# Explanation: The additive sequence is: 1, 99, 100, 199.
#              1 + 99 = 100, 99 + 100 = 199
#
#
#
# Constraints:
#
#
#       num consists only of digits '0'-'9'.
#       1 <= num.length <= 35
#
#
# Follow up:
#
# How would you handle overflow for very large input integers?# Time:  O(n^3)
# Space: O(n)


class Solution(object):
    def isAdditiveNumber(self, num):
        """
        :type num: str
        :rtype: bool
        """
        def add(a, b):
            res, carry, val = "", 0, 0
            for i in xrange(max(len(a), len(b))):
                val = carry
                if i < len(a):
                    val += int(a[-(i + 1)])
                if i < len(b):
                    val += int(b[-(i + 1)])
                carry, val = val / 10, val % 10
```

```python
            res += str(val)
        if carry:
            res += str(carry)
        return res[::-1]

    for i in xrange(1, len(num)):
        for j in xrange(i + 1, len(num)):
            s1, s2 = num[0:i], num[i:j]
            if (len(s1) > 1 and s1[0] == '0') or \
               (len(s2) > 1 and s2[0] == '0'):
                continue

            expected = add(s1, s2)
            cur = s1 + s2 + expected
            while len(cur) < len(num):
                s1, s2, expected = s2, expected, add(s2, expected)
                cur += expected
            if cur == num:
                return True
    return False
```

3sum.py

```python
# Given an array nums of n integers, are there elements a, b, c in nums 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.
#
# Example:
#
# Given array nums = [-1, 0, 1, 2, -1, -4],
#
# A solution set is:
# [
#   [-1, 0, 1],
#   [-1, -1, 2]
# ]# Time:  O(n^2)
# Space: O(1)

import collections


class Solution(object):
    def threeSum(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        nums, result, i = sorted(nums), [], 0
        while i < len(nums) - 2:
            if i == 0 or nums[i] != nums[i - 1]:
                j, k = i + 1, len(nums) - 1
                while j < k:
                    if nums[i] + nums[j] + nums[k] < 0:
                        j += 1
                    elif nums[i] + nums[j] + nums[k] > 0:
                        k -= 1
                    else:
                        result.append([nums[i], nums[j], nums[k]])
                        j, k = j + 1, k - 1
                        while j < k and nums[j] == nums[j - 1]:
                            j += 1
                        while j < k and nums[k] == nums[k + 1]:
                            k -= 1
            i += 1
        return result

    def threeSum2(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        d = collections.Counter(nums)
        nums_2 = [x[0] for x in d.items() if x[1] > 1]
        nums_new = sorted([x[0] for x in d.items()])
        rtn = [[0, 0, 0]] if d[0] >= 3 else []
        for i, j in enumerate(nums_new):
            if j <= 0:
```

```
        numss2 = nums_new[i + 1:]
        for x, y in enumerate(numss2):
            if 0 - j - y in [j, y] and 0 - j - y in nums_2:
                if sorted([j, y, 0 - j - y]) not in rtn:
                    rtn.append(sorted([j, y, 0 - j - y]))
            if 0 - j - y not in [j, y] and 0 - j - y in nums_new:
                if sorted([j, y, 0 - j - y]) not in rtn:
                    rtn.append(sorted([j, y, 0 - j - y]))
return rtn
```

minimum-cost-tree-from-leaf-values.py

```
# Given an array arr of positive integers, consider all binary trees such that:
#
#
#       Each node has either 0 or 2 children;
#       The values of arr correspond to the values of each leaf in an in-order
# traversal of the tree.  (Recall that a node is a leaf if and only if it has 0
# children.)
#       The value of each non-leaf node is equal to the product of the largest
# leaf value in its left and right subtree respectively.
#
#
# Among all possible binary trees considered, return the smallest possible sum
# of the values of each non-leaf node.  It is guaranteed this sum fits into a
# 32-bit integer.
#
#
# Example 1:
#
# Input: arr = [6,2,4]
# Output: 32
# Explanation:
# There are two possible trees.  The first has non-leaf node sum 36, and the
# second has non-leaf node sum 32.
#
#     24            24
#    /  \          /  \
#   12   4        6    8
#  /  \               / \
# 6    2             2   4
#
#
#
# Constraints:
#
#
#       2 <= arr.length <= 40
#       1 <= arr[i] <= 15
#       It is guaranteed that the answer fits into a 32-bit signed integer
# (ie. it is less than 2^31).# Time:  O(n)
# Space: O(n)

class Solution(object):
    def mctFromLeafValues(self, arr):
        """
        :type arr: List[int]
        :rtype: int
        """
        result = 0
        stk = [float("inf")]
        for x in arr:
            while stk[-1] <= x:
                result += stk.pop() * min(stk[-1], x)
            stk.append(x)
        while len(stk) > 2:
            result += stk.pop() * stk[-1]
        return result
```

longest-palindromic-subsequence.py

```python
# Given a string s, find the longest palindromic subsequence's length in s. You
# may assume that the maximum length of s is 1000.
#
# Example 1:
#
# Input:
#
# "bbbab"
#
# Output:
#
# 4
#
# One possible longest palindromic subsequence is "bbbb".
#
#
#
# Example 2:
#
# Input:
#
# "cbbd"
#
# Output:
#
# 2
#
# One possible longest palindromic subsequence is "bb".
#
# Constraints:
#
#
#       1 <= s.length <= 1000
#       s consists only of lowercase English letters.# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def longestPalindromeSubseq(self, s):
        """
        :type s: str
        :rtype: int
        """
        if s == s[::-1]:  # optional, to optimize special case
            return len(s)

        dp = [[1] * len(s) for _ in xrange(2)]
        for i in reversed(xrange(len(s))):
            for j in xrange(i+1, len(s)):
                if s[i] == s[j]:
                    dp[i%2][j] = 2 + dp[(i+1)%2][j-1] if i+1 <= j-1 else 2
                else:
                    dp[i%2][j] = max(dp[(i+1)%2][j], dp[i%2][j-1])
        return dp[0][-1]
```

closest-divisors.py

```python
# Given an integer num, find the closest two integers in absolute difference
# whose product equals num + 1 or num + 2.
#
# Return the two integers in any order.
#
#
# Example 1:
#
# Input: num = 8
# Output: [3,3]
# Explanation: For num + 1 = 9, the closest divisors are 3 & 3, for num + 2 =
# 10, the closest divisors are 2 & 5, hence 3 & 3 is chosen.
#
#
# Example 2:
#
# Input: num = 123
# Output: [5,25]
#
#
# Example 3:
#
# Input: num = 999
# Output: [40,25]
#
#
#
# Constraints:
#
#
#       1 <= num <= 10^9# Time:  O(sqrt(n))
# Space: O(1)

class Solution(object):
    def closestDivisors(self, num):
        """
        :type num: int
        :rtype: List[int]
        """
        def divisors(n):
            for d in reversed(xrange(1, int(n**0.5)+1)):
                if n % d == 0:
                    return d, n//d
            return 1, n

        return min([divisors(num+1), divisors(num+2)], key=lambda x: x[1]-x[0])



# Time:  O(sqrt(n))
# Space: O(1)
class Solution2(object):
    def closestDivisors(self, num):
        """
        :type num: int
        :rtype: List[int]
        """
        result, d = [1, num+1], 1
```

```python
while d*d <= num+2:
    if (num+2) % d == 0:
        result = [d, (num+2)//d]
    if (num+1) % d == 0:
        result = [d, (num+1)//d]
    d += 1
return result
```

minimum-moves-to-equal-array-elements-ii.py

```
# Given a non-empty integer array, find the minimum number of moves required to
# make all array elements equal, where a move is incrementing a selected element
# by 1 or decrementing a selected element by 1.
#
# You may assume the array's length is at most 10,000.
#
# Example:
# Input:
# [1,2,3]
#
# Output:
# 2
#
# Explanation:
# Only two moves are needed (remember each move increments or decrements one
# element):
#
# [1,2,3]  =>  [2,2,3]  =>  [2,2,2]# Time:  O(n) on average
# Space: O(1)

from random import randint

# Quick select solution.
class Solution(object):
    def minMoves2(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        def kthElement(nums, k):
            def PartitionAroundPivot(left, right, pivot_idx, nums):
                pivot_value = nums[pivot_idx]
                new_pivot_idx = left
                nums[pivot_idx], nums[right] = nums[right], nums[pivot_idx]
                for i in xrange(left, right):
                    if nums[i] > pivot_value:
                        nums[i], nums[new_pivot_idx] = nums[new_pivot_idx], nums[i]
                        new_pivot_idx += 1

                nums[right], nums[new_pivot_idx] = nums[new_pivot_idx], nums[right]
                return new_pivot_idx

            left, right = 0, len(nums) - 1
            while left <= right:
                pivot_idx = randint(left, right)
                new_pivot_idx = PartitionAroundPivot(left, right, pivot_idx, nums)
                if new_pivot_idx == k:
                    return nums[new_pivot_idx]
                elif new_pivot_idx > k:
                    right = new_pivot_idx - 1
                else:  # new_pivot_idx < k.
                    left = new_pivot_idx + 1

        median = kthElement(nums, len(nums)//2)
        return sum(abs(num - median) for num in nums)

    def minMoves22(self, nums):
        """
```

```
:type nums: List[int]
:rtype: int
"""
median = sorted(nums)[len(nums) / 2]
return sum(abs(num - median) for num in nums)
```

smallest-subsequence-of-distinct-characters.py

```python
# Return the lexicographically smallest subsequence of text that contains all
# the distinct characters of text exactly once.
#
# Example 1:
#
# Input: "cdadabcc"
# Output: "adbc"
#
#
#
# Example 2:
#
# Input: "abcd"
# Output: "abcd"
#
#
#
# Example 3:
#
# Input: "ecbacba"
# Output: "eacb"
#
#
#
# Example 4:
#
# Input: "leetcode"
# Output: "letcod"
#
#
#
#
# Constraints:
#
#
#       1 <= text.length <= 1000
#       text consists of lowercase English letters.
#
#
# Note: This question is the same as 316: https://leetcode.com/problems/remove-
# duplicate-letters/# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def smallestSubsequence(self, text):
        """
        :type text: str
        :rtype: str
        """
        count = collections.Counter(text)

        lookup, stk = set(), []
        for c in text:
            if c not in lookup:
                while stk and stk[-1] > c and count[stk[-1]]:
```

```
            lookup.remove(stk.pop())
        stk += c
        lookup.add(c)
    count[c] -= 1
return "".join(stk)
```

find-bottom-left-tree-value.py

```
# Given a binary tree, find the leftmost value in the last row of the tree.
#
#
# Example 1:
#
# Input:
#
#     2
#    / \
#   1   3
#
# Output:
# 1
#
#
#
#   Example 2:
#
# Input:
#
#         1
#        / \
#       2   3
#      /   / \
#     4   5   6
#        /
#       7
#
# Output:
# 7
#
#
#
# Note:
# You may assume the tree (i.e., the given root node) is not NULL.# Time:  O(n)
# Space: O(h)

class Solution(object):
    def findBottomLeftValue(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def findBottomLeftValueHelper(root, curr_depth, max_depth, bottom_left_value):
            if not root:
                return max_depth, bottom_left_value
            if not root.left and not root.right and curr_depth+1 > max_depth:
                return curr_depth+1, root.val
            max_depth, bottom_left_value = findBottomLeftValueHelper(root.left, curr_depth+1, max_depth, botto
            max_depth, bottom_left_value = findBottomLeftValueHelper(root.right, curr_depth+1, max_depth, bott
            return max_depth, bottom_left_value

        result, max_depth = 0, 0
        return findBottomLeftValueHelper(root, 0, max_depth, result)[1]


# Time:  O(n)
# Space: O(n)
```

```python
class Solution2(object):
    def findBottomLeftValue(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        last_node, queue = None, [root]
        while queue:
            last_node = queue.pop(0)
            queue.extend([n for n in [last_node.right, last_node.left] if n])
        return last_node.value
```

combinations.py

```python
# Given two integers n and k, return all possible combinations of k numbers out
# of 1 ... n.
#
# You may return the answer in any order.
#
#
# Example 1:
#
# Input: n = 4, k = 2
# Output:
# [
#   [2,4],
#   [3,4],
#   [2,3],
#   [1,2],
#   [1,3],
#   [1,4],
# ]
#
#
# Example 2:
#
# Input: n = 1, k = 1
# Output: [[1]]
#
#
#
# Constraints:
#
#
#       1 <= n <= 20
#       1 <= k <= n# Time:  O(k * C(n, k))
# Space: O(k)

class Solution(object):
    def combine(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: List[List[int]]
        """
        if k > n:
            return []
        nums, idxs = range(1, n+1), range(k)
        result = [[nums[i] for i in idxs]]
        while True:
            for i in reversed(xrange(k)):
                if idxs[i] != i+n-k:
                    break
            else:
                break
            idxs[i] += 1
            for j in xrange(i+1, k):
                idxs[j] = idxs[j-1]+1
            result.append([nums[i] for i in idxs])
        return result
```

```python
# Time:  O(k * C(n, k))
# Space: O(k)
class Solution2(object):
    def combine(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: List[List[int]]
        """
        result, combination = [], []
        i = 1
        while True:
            if len(combination) == k:
                result.append(combination[:])
            if len(combination) == k or \
               len(combination)+(n-i+1) < k:
                if not combination:
                    break
                i = combination.pop()+1
            else:
                combination.append(i)
                i += 1
        return result


# Time:  O(k * C(n, k))
# Space: O(k)
class Solution3(object):
    def combine(self, n, k):
        """
        :type n: int
        :type k: int
        :rtype: List[List[int]]
        """
        def combineDFS(n, start, intermediate, k, result):
            if k == 0:
                result.append(intermediate[:])
                return
            for i in xrange(start, n):
                intermediate.append(i+1)
                combineDFS(n, i+1, intermediate, k-1, result)
                intermediate.pop()

        result = []
        combineDFS(n, 0, [], k, result)
        return result
```

capacity-to-ship-packages-within-d-days.py

```
# A conveyor belt has packages that must be shipped from one port to another
# within D days.
#
# The i-th package on the conveyor belt has a weight of weights[i].  Each day,
# we load the ship with packages on the conveyor belt (in the order given by
# weights). We may not load more weight than the maximum weight capacity of the
# ship.
#
# Return the least weight capacity of the ship that will result in all the
# packages on the conveyor belt being shipped within D days.
#
#
#
# Example 1:
#
# Input: weights = [1,2,3,4,5,6,7,8,9,10], D = 5
# Output: 15
# Explanation:
# A ship capacity of 15 is the minimum to ship all the packages in 5 days like
# this:
# 1st day: 1, 2, 3, 4, 5
# 2nd day: 6, 7
# 3rd day: 8
# 4th day: 9
# 5th day: 10
#
# Note that the cargo must be shipped in the order given, so using a ship of
# capacity 14 and splitting the packages into parts like (2, 3, 4, 5), (1, 6, 7),
# (8), (9), (10) is not allowed.
#
#
# Example 2:
#
# Input: weights = [3,2,2,4,1,4], D = 3
# Output: 6
# Explanation:
# A ship capacity of 6 is the minimum to ship all the packages in 3 days like
# this:
# 1st day: 3, 2
# 2nd day: 2, 4
# 3rd day: 1, 4
#
#
# Example 3:
#
# Input: weights = [1,2,3,1,1], D = 4
# Output: 3
# Explanation:
# 1st day: 1
# 2nd day: 2
# 3rd day: 3
# 4th day: 1, 1
#
#
#
# Constraints:
#
#
```

```python
#       1 <= D <= weights.length <= 50000
#       1 <= weights[i] <= 500# Time:  O(nlogr)
# Space: O(1)

class Solution(object):
    def shipWithinDays(self, weights, D):
        """
        :type weights: List[int]
        :type D: int
        :rtype: int
        """
        def possible(weights, D, mid):
            result, curr = 1, 0
            for w in weights:
                if curr+w > mid:
                    result += 1
                    curr = 0
                curr += w
            return result <= D

        left, right = max(weights), sum(weights)
        while left <= right:
            mid = left + (right-left)//2
            if possible(weights, D, mid):
                right = mid-1
            else:
                left = mid+1
        return left
```

game-of-life.py

```python
# 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):
#
#
#       Any live cell with fewer than two live neighbors dies, as if caused by
# under-population.
#       Any live cell with two or three live neighbors lives on to the next
# generation.
#       Any live cell with more than three live neighbors dies, as if by over-
# population..
#       Any dead cell with exactly three live neighbors becomes a live cell, as
# if by reproduction.
#
#
# Write a function to compute the next state (after one update) of the board
# given its current state. The next state is created by applying the above rules
# simultaneously to every cell in the current state, where births and deaths occur
# simultaneously.
#
# Example:
#
# Input:
# [
#   [0,1,0],
#   [0,0,1],
#   [1,1,1],
#   [0,0,0]
# ]
# Output:
# [
#   [0,0,0],
#   [1,0,1],
#   [0,1,1],
#   [0,1,0]
# ]
#
#
# Follow up:
#
#
#       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.
#       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?# Time:  O(m * n)
# Space: O(1)

class Solution(object):
    def gameOfLife(self, board):
        """
        :type board: List[List[int]]
```

```
        :rtype: void Do not return anything, modify board in-place instead.
        """
        m = len(board)
        n = len(board[0]) if m else 0
        for i in xrange(m):
            for j in xrange(n):
                count = 0
                ## Count live cells in 3x3 block.
                for I in xrange(max(i-1, 0), min(i+2, m)):
                    for J in xrange(max(j-1, 0), min(j+2, n)):
                        count += board[I][J] & 1

                # if (count == 4 && board[i][j]) means:
                #     Any live cell with three live neighbors lives.
                # if (count == 3) means:
                #     Any live cell with two live neighbors.
                #     Any dead cell with exactly three live neighbors lives.
                if (count == 4 and board[i][j]) or count == 3:
                    board[i][j] |= 2  # Mark as live.

        for i in xrange(m):
            for j in xrange(n):
                board[i][j] >>= 1  # Update to the next state.
```

online-stock-span.py

```python
# Example 1:
#
# Input: ["StockSpanner","next","next","next","next","next","next","next"],
# [[],[100],[80],[60],[70],[60],[75],[85]]
# Output: [null,1,1,1,2,1,4,6]
# Explanation:
# First, S = StockSpanner() is initialized.  Then:
# S.next(100) is called and returns 1,
# S.next(80) is called and returns 1,
# S.next(60) is called and returns 1,
# S.next(70) is called and returns 2,
# S.next(60) is called and returns 1,
# S.next(75) is called and returns 4,
# S.next(85) is called and returns 6.
#
# Note that (for example) S.next(75) returned 4, because the last 4 prices
# (including today's price of 75) were less than or equal to today's price.
#
#
#
#
# Note:
#
#
#       Calls to StockSpanner.next(int price) will have 1 <= price <= 10^5.
#       There will be at most 10000 calls to StockSpanner.next per test case.
#       There will be at most 150000 calls to StockSpanner.next across all test
# cases.
#       The total time limit for this problem has been reduced by 75% for C++,
# and 50% for all other languages.# Time:  O(n)
# Space: O(n)

class StockSpanner(object):

    def __init__(self):
        self.__s = []

    def next(self, price):
        """
        :type price: int
        :rtype: int
        """
        result = 1
        while self.__s and self.__s[-1][0] <= price:
            result += self.__s.pop()[1]
        self.__s.append([price, result])
        return result
```

interval-list-intersections.py

```python
# Given two lists of closed intervals, each list of intervals is pairwise
# disjoint and in sorted order.
#
# Return the intersection of these two interval lists.
#
# (Formally, a closed interval [a, b] (with a <= b) denotes the set of real
# numbers x with a <= x <= b.  The intersection of two closed intervals is a set
# of real numbers that is either empty, or can be represented as a closed
# interval.  For example, the intersection of [1, 3] and [2, 4] is [2, 3].)
#
#
#
#
# Example 1:
#
#
#
# Input: A = [[0,2],[5,10],[13,23],[24,25]], B = [[1,5],[8,12],[15,24],[25,26]]
# Output: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
#
#
#
#
# Note:
#
#
#       0 <= A.length < 1000
#       0 <= B.length < 1000
#       0 <= A[i].start, A[i].end, B[i].start, B[i].end < 10^9# Time:  O(m + n)
# Space: O(1)

# Definition for an interval.
class Interval(object):
    def __init__(self, s=0, e=0):
        self.start = s
        self.end = e


class Solution(object):
    def intervalIntersection(self, A, B):
        """
        :type A: List[Interval]
        :type B: List[Interval]
        :rtype: List[Interval]
        """
        result = []
        i, j = 0, 0
        while i < len(A) and j < len(B):
            left = max(A[i].start, B[j].start)
            right = min(A[i].end, B[j].end)
            if left <= right:
                result.append(Interval(left, right))
            if A[i].end < B[j].end:
                i += 1
            else:
                j += 1
        return result
```

lexicographical-numbers.py

```python
# Given an integer n, return 1 - n in lexicographical order.
#
# For example, given 13, return: [1,10,11,12,13,2,3,4,5,6,7,8,9].
#
# Please optimize your algorithm to use less time and space. The input size may
# be as large as 5,000,000.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def lexicalOrder(self, n):
        result = []

        i = 1
        while len(result) < n:
            k = 0
            while i * 10**k <= n:
                result.append(i * 10**k)
                k += 1

            num = result[-1] + 1
            while num <= n and num % 10:
                result.append(num)
                num += 1

            if not num % 10:
                num -= 1
            else:
                num /= 10

            while num % 10 == 9:
                num /= 10

            i = num+1

        return result
```

most-frequent-subtree-sum.py

```python
# Given the root of a tree, you are asked to find the most frequent subtree sum.
# The subtree sum of a node is defined as the sum of all the node values formed by
# the subtree rooted at that node (including the node itself). So what is the most
# frequent subtree sum value? If there is a tie, return all the values with the
# highest frequency in any order.
#
#
# Examples 1
#
# Input:
#   5
#  /  \
# 2   -3
#
# return [2, -3, 4], since all the values happen only once, return all of them
# in any order.
#
#
# Examples 2
#
# Input:
#   5
#  /  \
# 2   -5
#
# return [2], since 2 happens twice, however -5 only occur once.
#
#
# Note:
# You may assume the sum of values in any subtree is in the range of 32-bit
# signed integer.# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def findFrequentTreeSum(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        def countSubtreeSumHelper(root, counts):
            if not root:
                return 0
            total = root.val + \
                    countSubtreeSumHelper(root.left, counts) + \
                    countSubtreeSumHelper(root.right, counts)
            counts[total] += 1
            return total

        counts = collections.defaultdict(int)
        countSubtreeSumHelper(root, counts)
        max_count = max(counts.values()) if counts else 0
        return [total for total, count in counts.iteritems() if count == max_count]
```

k-closest-points-to-origin.py

```python
# Example 1:
#
# Input: points = [[1,3],[-2,2]], K = 1
# Output: [[-2,2]]
# Explanation:
# The distance between (1, 3) and the origin is sqrt(10).
# The distance between (-2, 2) and the origin is sqrt(8).
# Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.
# We only want the closest K = 1 points from the origin, so the answer is just
# [[-2,2]].
#
#
#
# Example 2:
#
# Input: points = [[3,3],[5,-1],[-2,4]], K = 2
# Output: [[3,3],[-2,4]]
# (The answer [[-2,4],[3,3]] would also be accepted.)
#
#
#
#
# Note:
#
#
#       1 <= K <= points.length <= 10000
#       -10000 < points[i][0] < 10000
#       -10000 < points[i][1] < 10000# Time:  O(n) on average
# Space: O(1)

# quick select solution
from random import randint


class Solution(object):
    def kClosest(self, points, K):
        """
        :type points: List[List[int]]
        :type K: int
        :rtype: List[List[int]]
        """
        def dist(point):
            return point[0]**2 + point[1]**2

        def kthElement(nums, k, compare):
            def PartitionAroundPivot(left, right, pivot_idx, nums, compare):
                new_pivot_idx = left
                nums[pivot_idx], nums[right] = nums[right], nums[pivot_idx]
                for i in xrange(left, right):
                    if compare(nums[i], nums[right]):
                        nums[i], nums[new_pivot_idx] = nums[new_pivot_idx], nums[i]
                        new_pivot_idx += 1

                nums[right], nums[new_pivot_idx] = nums[new_pivot_idx], nums[right]
                return new_pivot_idx

            left, right = 0, len(nums) - 1
            while left <= right:
```

```python
            pivot_idx = randint(left, right)
            new_pivot_idx = PartitionAroundPivot(left, right, pivot_idx, nums, compare)
            if new_pivot_idx == k:
                return
            elif new_pivot_idx > k:
                right = new_pivot_idx - 1
            else:  # new_pivot_idx < k.
                left = new_pivot_idx + 1

        kthElement(points, K, lambda a, b: dist(a) < dist(b))
        return points[:K]


# Time:  O(nlogk)
# Space: O(k)
import heapq


class Solution2(object):
    def kClosest(self, points, K):
        """
        :type points: List[List[int]]
        :type K: int
        :rtype: List[List[int]]
        """
        def dist(point):
            return point[0]**2 + point[1]**2

        max_heap = []
        for point in points:
            heapq.heappush(max_heap, (-dist(point), point))
            if len(max_heap) > K:
                heapq.heappop(max_heap)
        return [heapq.heappop(max_heap)[1] for _ in xrange(len(max_heap))]
```

implement-magic-dictionary.py

```python
# Design a data structure which is initialized with a list of different words.
# After that, we will give you a string, and you should find out if you can change
# exactly one character in this string to match any word in the data structure.
#
# Implement the MagicDictionary class:
#
#
#       MagicDictionary() Initializes the object.
#       void buildDict(String[] dictionary) Sets the data structure with an
# array of distinct strings dictionary.
#       bool search(String searchWord) Returns true if you can change exactly
# one character in word to match any string in the data structure, otherwise
# returns false.
#
#
#
# Example 1:
#
# Input
# ["MagicDictionary", "buildDict", "search", "search", "search", "search"]
# [[], [["hello", "leetcode"]], ["hello"], ["hhllo"], ["hell"], ["leetcoded"]]
# Output
# [null, null, false, true, false, false]
#
# Explanation
# MagicDictionary magicDictionary = new MagicDictionary();
# magicDictionary.buildDict(["hello", "leetcode"]);
# magicDictionary.search("hello"); // return False
# magicDictionary.search("hhllo"); // We can change the second 'h' to 'e' to
# match "hello" so we return True
# magicDictionary.search("hell"); // return False
# magicDictionary.search("leetcoded"); // return False
#
#
#
# Constraints:
#
#
#       1 <= dictionary.length <= 100
#       1 <= dictionary[i],length <= 100
#       dictionary[i] consist of only lower-case English letters.
#       All the strings in dictionary are distinct.
#       1 <= searchWord.length <= 100
#       searchWord consist of only lower-case English letters.
#       buildDict will be called only one time.
#       At most 100 calls will be made to search,# Time:  O(n), n is the length of the word
# Space: O(d)

import collections


class MagicDictionary(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
        _trie = lambda: collections.defaultdict(_trie)
```

```python
        self.trie = _trie()


    def buildDict(self, dictionary):
        """
        Build a dictionary through a list of words
        :type dictionary: List[str]
        :rtype: void
        """
        for word in dictionary:
            reduce(dict.__getitem__, word, self.trie).setdefault("_end")


    def search(self, word):
        """
        Returns if there is any word in the trie that equals to the given word after modifying exactly one cha
        :type word: str
        :rtype: bool
        """
        def find(word, curr, i, mistakeAllowed):
            if i == len(word):
                return "_end" in curr and not mistakeAllowed

            if word[i] not in curr:
                return any(find(word, curr[c], i+1, False) for c in curr if c != "_end") \
                       if mistakeAllowed else False

            if mistakeAllowed:
                return find(word, curr[word[i]], i+1, True) or \
                       any(find(word, curr[c], i+1, False) \
                           for c in curr if c not in ("_end", word[i]))
            return find(word, curr[word[i]], i+1, False)

        return find(word, self.trie, 0, True)
```

h-index.py

```python
# 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."
#
# Example:
#
# Input: citations = [3,0,6,1,5]
# Output: 3
# Explanation: [3,0,6,1,5] 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, her h-index is 3.
#
# Note: If there are several possible values for h, the maximum one is taken as
# the h-index.# Time:  O(n)
# Space: O(n)

class Solution(object):
    def hIndex(self, citations):
        """
        :type citations: List[int]
        :rtype: int
        """
        n = len(citations)
        count = [0] * (n + 1)
        for x in citations:
            # Put all x >= n in the same bucket.
            if x >= n:
                count[n] += 1
            else:
                count[x] += 1

        h = 0
        for i in reversed(xrange(0, n + 1)):
            h += count[i]
            if h >= i:
                return i
        return h

# Time:  O(nlogn)
# Space: O(1)
class Solution2(object):
    def hIndex(self, citations):
        """
        :type citations: List[int]
        :rtype: int
        """
        citations.sort(reverse=True)
        h = 0
        for x in citations:
            if x >= h + 1:
                h += 1
            else:
```

```python
                break
        return h

# Time:  O(nlogn)
# Space: O(n)
class Solution3(object):
    def hIndex(self, citations):
        """
        :type citations: List[int]
        :rtype: int
        """
        return sum(x >= i + 1 for i, x in enumerate(sorted(citations, reverse=True)))
```

construct-binary-tree-from-preorder-and-postorder-traversal.py

```
# Example 1:
#
# Input: pre = [1,2,4,5,3,6,7], post = [4,5,2,6,7,3,1]
# Output: [1,2,3,4,5,6,7]
#
#
#
#
# Note:
#
#
#       1 <= pre.length == post.length <= 30
#       pre[] and post[] are both permutations of 1, 2, ..., pre.length.
#       It is guaranteed an answer exists. If there exists multiple answers, you
# can return any of them.# Time:  O(n)
# Space: O(h)

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution(object):
    def constructFromPrePost(self, pre, post):
        """
        :type pre: List[int]
        :type post: List[int]
        :rtype: TreeNode
        """
        stack = [TreeNode(pre[0])]
        j = 0
        for i in xrange(1, len(pre)):
            node = TreeNode(pre[i])
            while stack[-1].val == post[j]:
                stack.pop()
                j += 1
            if not stack[-1].left:
                stack[-1].left = node
            else:
                stack[-1].right = node
            stack.append(node)
        return stack[0]


# Time:  O(n)
# Space: O(n)
class Solution2(object):
    def constructFromPrePost(self, pre, post):
        """
        :type pre: List[int]
        :type post: List[int]
        :rtype: TreeNode
        """
        def constructFromPrePostHelper(pre, pre_s, pre_e, post, post_s, post_e, post_entry_idx_map):
            if pre_s >= pre_e or post_s >= post_e:
                return None
```

```python
    node = TreeNode(pre[pre_s])
    if pre_e-pre_s > 1:
        left_tree_size = post_entry_idx_map[pre[pre_s+1]]-post_s+1
        node.left = constructFromPrePostHelper(pre, pre_s+1, pre_s+1+left_tree_size,
                                               post, post_s, post_s+left_tree_size,
                                               post_entry_idx_map)
        node.right = constructFromPrePostHelper(pre, pre_s+1+left_tree_size, pre_e,
                                                post, post_s+left_tree_size, post_e-1,
                                                post_entry_idx_map)
    return node

post_entry_idx_map = {}
for i, val in enumerate(post):
    post_entry_idx_map[val] = i
return constructFromPrePostHelper(pre, 0, len(pre), post, 0, len(post), post_entry_idx_map)
```

integer-break.py

```python
# Example 2:
#
# Input: 10
# Output: 36
# Explanation: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36.
#
# Note: You may assume that n is not less than 2 and not larger than 58.# Time:  O(logn), pow is O(logn).
# Space: O(1)

class Solution(object):
    def integerBreak(self, n):
        """
        :type n: int
        :rtype: int
        """
        if n < 4:
            return n - 1

        #  Proof.
        #  1. Let n = a1 + a2 + ... + ak, product = a1 * a2 * ... * ak
        #      - For each ai >= 4, we can always maximize the product by:
        #        ai <= 2 * (ai - 2)
        #      - For each aj >= 5, we can always maximize the product by:
        #        aj <= 3 * (aj - 3)
        #
        #     Conclusion 1:
        #      - For n >= 4, the max of the product must be in the form of
        #        3^a * 2^b, s.t. 3a + 2b = n
        #
        #  2. To maximize the product = 3^a * 2^b s.t. 3a + 2b = n
        #      - For each b >= 3, we can always maximize the product by:
        #        3^a * 2^b <= 3^(a+2) * 2^(b-3) s.t. 3(a+2) + 2(b-3) = n
        #
        #     Conclusion 2:
        #      - For n >= 4, the max of the product must be in the form of
        #        3^Q * 2^R, 0 <= R < 3 s.t. 3Q + 2R = n
        #        i.e.
        #          if n = 3Q + 0,   the max of the product = 3^Q * 2^0
        #          if n = 3Q + 2,   the max of the product = 3^Q * 2^1
        #          if n = 3Q + 2*2, the max of the product = 3^Q * 2^2

        res = 0
        if n % 3 == 0:            #  n = 3Q + 0, the max is 3^Q * 2^0
            res = 3 ** (n // 3)
        elif n % 3 == 2:          #  n = 3Q + 2, the max is 3^Q * 2^1
            res = 3 ** (n // 3) * 2
        else:                     #  n = 3Q + 4, the max is 3^Q * 2^2
            res = 3 ** (n // 3 - 1) * 4
        return res


# Time:  O(n)
# Space: O(1)
# DP solution.
class Solution2(object):
    def integerBreak(self, n):
        """
        :type n: int
```

```
:rtype: int
"""
if n < 4:
    return n - 1

# integerBreak(n) = max(integerBreak(n - 2) * 2, integerBreak(n - 3) * 3)
res = [0, 1, 2, 3]
for i in xrange(4, n + 1):
    res[i % 4] = max(res[(i - 2) % 4] * 2, res[(i - 3) % 4] * 3)
return res[n % 4]
```

predict-the-winner.py

```
# Given an array of scores that are non-negative integers. Player 1 picks one of
# the numbers from either end of the array followed by the player 2 and then
# player 1 and so on. Each time a player picks a number, that number will not be
# available for the next player. This continues until all the scores have been
# chosen. The player with the maximum score wins.
#
# Given an array of scores, predict whether player 1 is the winner. You can
# assume each player plays to maximize his score.
#
# Example 1:
#
# Input: [1, 5, 2]
# Output: False
# Explanation: Initially, player 1 can choose between 1 and 2.
# If he chooses 2 (or 1), then player 2 can choose from 1 (or 2) and 5. If
# player 2 chooses 5, then player 1 will be left with 1 (or 2).
# So, final score of player 1 is 1 + 2 = 3, and player 2 is 5.
# Hence, player 1 will never be the winner and you need to return False.
#
#
#
#
# Example 2:
#
# Input: [1, 5, 233, 7]
# Output: True
# Explanation: Player 1 first chooses 1. Then player 2 have to choose between 5
# and 7. No matter which number player 2 choose, player 1 can choose 233.
# Finally, player 1 has more score (234) than player 2 (12), so you need to
# return True representing player1 can win.
#
#
#
# Constraints:
#
#
#       1 <= length of the array <= 20.
#       Any scores in the given array are non-negative integers and will not
# exceed 10,000,000.
#       If the scores of both players are equal, then player 1 is still the
# winner.# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def PredictTheWinner(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        if len(nums) % 2 == 0 or len(nums) == 1:
            return True

        dp = [0] * len(nums)
        for i in reversed(xrange(len(nums))):
            dp[i] = nums[i]
            for j in xrange(i+1, len(nums)):
                dp[j] = max(nums[i] - dp[j], nums[j] - dp[j - 1])
```

```
        return dp[-1] >= 0
```

remove-covered-intervals.py

```python
# Given a list of intervals, remove all intervals that are covered by another
# interval in the list. Interval [a,b) is covered by interval [c,d) if and only if
# c <= a and b <= d.
#
# After doing so, return the number of remaining intervals.
#
#
# Example 1:
#
# Input: intervals = [[1,4],[3,6],[2,8]]
# Output: 2
# Explanation: Interval [3,6] is covered by [2,8], therefore it is removed.
#
#
#
# Constraints:
#
#
#       1 <= intervals.length <= 1000
#       0 <= intervals[i][0] < intervals[i][1] <= 10^5
#       intervals[i] != intervals[j] for all i != j# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def removeCoveredIntervals(self, intervals):
        """
        :type intervals: List[List[int]]
        :rtype: int
        """
        intervals.sort(key=lambda x: [x[0], -x[1]])
        result, max_right = 0, 0
        for left, right in intervals:
            result += int(right > max_right)
            max_right = max(max_right, right)
        return result
```

find-peak-element.py

```
# A peak element is an element that is greater than its neighbors.
#
# Given an input array nums, where nums[i]  nums[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 nums[-1] = nums[n] = -.
#
# Example 1:
#
# Input: nums = [1,2,3,1]
# Output: 2
# Explanation: 3 is a peak element and your function should return the index
# number 2.
#
# Example 2:
#
# Input: nums = [1,2,1,3,5,6,4]
# Output: 1 or 5
# Explanation: Your function can return either index number 1 where the peak
# element is 2,
#              or index number 5 where the peak element is 6.
#
#
# Follow up: Your solution should be in logarithmic complexity.# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def findPeakElement(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        left, right = 0, len(nums) - 1

        while left < right:
            mid = left + (right - left) / 2
            if nums[mid] > nums[mid + 1]:
                right = mid
            else:
                left = mid + 1

        return left
```

reorder-list.py

```python
# Given a singly linked list L: L0→L1→...→Ln-1→Ln,
#
# reorder it to: L0→Ln→L1→Ln-1→L2→Ln-2→...
#
# You may not modify the values in the list's nodes, only nodes itself may be
# changed.
#
# Example 1:
#
# Given 1->2->3->4, reorder it to 1->4->2->3.
#
# Example 2:
#
# Given 1->2->3->4->5, reorder it to 1->5->2->4->3.# Time:  O(n)
# Space: O(1)

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

    def __repr__(self):
        if self:
            return "{} -> {}".format(self.val, repr(self.next))

class Solution(object):
    # @param head, a ListNode
    # @return nothing
    def reorderList(self, head):
        if head == None or head.next == None:
            return head

        fast, slow, prev = head, head, None
        while fast != None and fast.next != None:
            fast, slow, prev = fast.next.next, slow.next, slow
        current, prev.next, prev = slow, None, None

        while current != None:
            current.next, prev, current = prev, current, current.next

        l1, l2 = head, prev
        dummy = ListNode(0)
        current = dummy

        while l1 != None and l2 != None:
            current.next, current, l1 = l1, l1, l1.next
            current.next, current, l2 = l2, l2, l2.next

        return dummy.next
```

maximum-length-of-pair-chain.py

```python
# You are given n pairs of numbers. In every pair, the first number is always
# smaller than the second number.
#
#
#
# Now, we define a pair (c, d) can follow another pair (a, b) if and only if b <
# c. Chain of pairs can be formed in this fashion.
#
#
#
# Given a set of pairs, find the length longest chain which can be formed. You
# needn't use up all the given pairs. You can select pairs in any order.
#
#
#
# Example 1:
#
# Input: [[1,2], [2,3], [3,4]]
# Output: 2
# Explanation: The longest chain is [1,2] -> [3,4]
#
#
#
# Note:
#
#
# The number of given pairs will be in the range [1, 1000].# Time:  O(nlogn)
# Space: O(1)

class Solution(object):
    def findLongestChain(self, pairs):
        """
        :type pairs: List[List[int]]
        :rtype: int
        """
        pairs.sort(key=lambda x: x[1])
        cnt, i = 0, 0
        for j in xrange(len(pairs)):
            if j == 0 or pairs[i][1] < pairs[j][0]:
                cnt += 1
                i = j
        return cnt
```

integer-to-roman.py

```
# Roman numerals are represented by seven different symbols: I, V, X, L, C, D
# and M.
#
# Symbol       Value
# I             1
# V             5
# X             10
# L             50
# C             100
# D             500
# M             1000
#
# For example, two is written as II in Roman numeral, just two one's added
# together. Twelve is written as, XII, which is simply X + II. The number twenty
# seven is written as XXVII, which is XX + V + II.
#
# Roman numerals are usually written largest to smallest from left to right.
# However, the numeral for four is not IIII. Instead, the number four is written
# as IV. Because the one is before the five we subtract it making four. The same
# principle applies to the number nine, which is written as IX. There are six
# instances where subtraction is used:
#
#
#       I can be placed before V (5) and X (10) to make 4 and 9.
#       X can be placed before L (50) and C (100) to make 40 and 90.
#       C can be placed before D (500) and M (1000) to make 400 and 900.
#
#
# Given an integer, convert it to a roman numeral. Input is guaranteed to be
# within the range from 1 to 3999.
#
# Example 1:
#
# Input: 3
# Output: "III"
#
# Example 2:
#
# Input: 4
# Output: "IV"
#
# Example 3:
#
# Input: 9
# Output: "IX"
#
# Example 4:
#
# Input: 58
# Output: "LVIII"
# Explanation: L = 50, V = 5, III = 3.
#
#
# Example 5:
#
# Input: 1994
# Output: "MCMXCIV"
# Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.# Time:  O(n)
```

```
# Space: O(1)

class Solution(object):
    def intToRoman(self, num):
        """
        :type num: int
        :rtype: str
        """
        numeral_map = {1: "I", 4: "IV", 5: "V", 9: "IX", \
                       10: "X", 40: "XL", 50: "L", 90: "XC", \
                       100: "C", 400: "CD", 500: "D", 900: "CM", \
                       1000: "M"}
        keyset, result = sorted(numeral_map.keys()), []

        while num > 0:
            for key in reversed(keyset):
                while num / key > 0:
                    num -= key
                    result += numeral_map[key]

        return "".join(result)
```

rotting-oranges.py

```python
# Example 1:
#
#
#
# Input: [[2,1,1],[1,1,0],[0,1,1]]
# Output: 4
#
#
#
# Example 2:
#
# Input: [[2,1,1],[0,1,1],[1,0,1]]
# Output: -1
# Explanation:  The orange in the bottom left corner (row 2, column 0) is never
# rotten, because rotting only happens 4-directionally.
#
#
#
# Example 3:
#
# Input: [[0,2]]
# Output: 0
# Explanation:  Since there are already no fresh oranges at minute 0, the answer
# is just 0.
#
#
#
#
# Note:
#
#
#       1 <= grid.length <= 10
#       1 <= grid[0].length <= 10
#       grid[i][j] is only 0, 1, or 2.# Time:  O(m * n)
# Space: O(m * n)

import collections


class Solution(object):
    def orangesRotting(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        count = 0
        q = collections.deque()
        for r, row in enumerate(grid):
            for c, val in enumerate(row):
                if val == 2:
                    q.append((r, c, 0))
                elif val == 1:
                    count += 1

        result = 0
        while q:
```

```
        r, c, result = q.popleft()
        for d in directions:
            nr, nc = r+d[0], c+d[1]
            if not (0 <= nr < len(grid) and \
                    0 <= nc < len(grid[r])):
                continue
            if grid[nr][nc] == 1:
                count -= 1
                grid[nr][nc] = 2
                q.append((nr, nc, result+1))
    return result if count == 0 else -1
```

snapshot-array.py

```python
# Implement a SnapshotArray that supports the following interface:
#
#
#       SnapshotArray(int length) initializes an array-like data structure with
# the given length.  Initially, each element equals 0.
#       void set(index, val) sets the element at the given index to be equal to
# val.
#       int snap() takes a snapshot of the array and returns the snap_id: the
# total number of times we called snap() minus 1.
#       int get(index, snap_id) returns the value at the given index, at the
# time we took the snapshot with the given snap_id
#
#
#
# Example 1:
#
# Input: ["SnapshotArray","set","snap","set","get"]
# [[3],[0,5],[],[0,6],[0,0]]
# Output: [null,null,0,null,5]
# Explanation:
# SnapshotArray snapshotArr = new SnapshotArray(3); // set the length to be 3
# snapshotArr.set(0,5);  // Set array[0] = 5
# snapshotArr.snap();  // Take a snapshot, return snap_id = 0
# snapshotArr.set(0,6);
# snapshotArr.get(0,0);  // Get the value of array[0] with snap_id = 0, return 5
#
#
# Constraints:
#
#
#       1 <= length <= 50000
#       At most 50000 calls will be made to set, snap, and get.
#       0 <= index < length
#       0 <= snap_id < (the total number of times we call snap())
#       0 <= val <= 10^9# Time:  set: O(1)
#        get: O(logn), n is the total number of set
# Space: O(n)

import collections
import bisect


class SnapshotArray(object):

    def __init__(self, length):
        """
        :type length: int
        """
        self.__A = collections.defaultdict(lambda: [(-1, 0)])
        self.__snap_id = 0


    def set(self, index, val):
        """
        :type index: int
        :type val: int
        :rtype: None
        """
```

```python
        self.__A[index].append((self.__snap_id, val))


    def snap(self):
        """
        :rtype: int
        """
        self.__snap_id += 1
        return self.__snap_id - 1


    def get(self, index, snap_id):
        """
        :type index: int
        :type snap_id: int
        :rtype: int
        """
        i = bisect.bisect_right(self.__A[index], (snap_id+1, 0)) - 1
        return self.__A[index][i][1]
```

longest-word-in-dictionary-through-deleting.py

```python
# Given a string and a string dictionary, find the longest string in the
# dictionary that can be formed by deleting some characters of the given string.
# If there are more than one possible results, return the longest word with the
# smallest lexicographical order. If there is no possible result, return the empty
# string.
#
# Example 1:
#
# Input:
# s = "abpcplea", d = ["ale","apple","monkey","plea"]
#
# Output:
# "apple"
#
#
#
#
# Example 2:
#
# Input:
# s = "abpcplea", d = ["a","b","c"]
#
# Output:
# "a"
#
#
#
# Note:
#
#
# All the strings in the input will only contain lower-case letters.
# The size of the dictionary won't exceed 1,000.
# The length of all the strings in the input won't exceed 1,000.# Time:  O((d * l) * logd), l is the average l
# Space: O(1)

class Solution(object):
    def findLongestWord(self, s, d):
        """
        :type s: str
        :type d: List[str]
        :rtype: str
        """
        d.sort(key = lambda x: (-len(x), x))
        for word in d:
            i = 0
            for c in s:
                if i < len(word) and word[i] == c:
                    i += 1
            if i == len(word):
                return word
        return ""
```

reconstruct-itinerary.py

```
# 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:
#
#
#       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"].
#       All airports are represented by three capital letters (IATA code).
#       You may assume all tickets form at least one valid itinerary.
#       One must use all the tickets once and only once.
#
#
# Example 1:
#
# Input: [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]
# Output: ["JFK", "MUC", "LHR", "SFO", "SJC"]
#
#
# Example 2:
#
# Input: [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]
# Output: ["JFK","ATL","JFK","SFO","ATL","SFO"]
# Explanation: Another possible reconstruction is
# ["JFK","SFO","ATL","JFK","ATL","SFO"].
#              But it is larger in lexical order.# Time:  O(t! / (n1! * n2! * ... nk!)), t is the total number
#                                     ni is the number of the ticket which from is city i,
#                                     k is the total number of cities.
# Space: O(t)

import collections


class Solution(object):
    def findItinerary(self, tickets):
        """
        :type tickets: List[List[str]]
        :rtype: List[str]
        """
        def route_helper(origin, ticket_cnt, graph, ans):
            if ticket_cnt == 0:
                return True

            for i, (dest, valid)  in enumerate(graph[origin]):
                if valid:
                    graph[origin][i][1] = False
                    ans.append(dest)
                    if route_helper(dest, ticket_cnt - 1, graph, ans):
                        return ans
                    ans.pop()
                    graph[origin][i][1] = True
            return False

        graph = collections.defaultdict(list)
        for ticket in tickets:
            graph[ticket[0]].append([ticket[1], True])
```

```
for k in graph.keys():
    graph[k].sort()

origin = "JFK"
ans = [origin]
route_helper(origin, len(tickets), graph, ans)
return ans
```

add-two-numbers.py

```python
# You are given two non-empty linked lists representing two non-negative
# integers. 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.
#
# You may assume the two numbers do not contain any leading zero, except the
# number 0 itself.
#
# Example:
#
# Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
# Output: 7 -> 0 -> 8
# Explanation: 342 + 465 = 807.# Time:  O(n)
# Space: O(1)

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None


class Solution(object):
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        dummy = ListNode(0)
        current, carry = dummy, 0

        while l1 or l2:
            val = carry
            if l1:
                val += l1.val
                l1 = l1.next
            if l2:
                val += l2.val
                l2 = l2.next
            carry, val = divmod(val, 10)
            current.next = ListNode(val)
            current = current.next

        if carry == 1:
            current.next = ListNode(1)

        return dummy.next
```

prison-cells-after-n-days.py

```python
# Example 2:
#
# Input: cells = [1,0,0,1,0,0,1,0], N = 1000000000
# Output: [0,0,1,1,1,1,1,0]
#
#
#
#
# Note:
#
#
#       cells.length == 8
#       cells[i] is in {0, 1}
#       1 <= N <= 10^9# Time:  O(1)
# Space: O(1)

class Solution(object):
    def prisonAfterNDays(self, cells, N):
        """
        :type cells: List[int]
        :type N: int
        :rtype: List[int]
        """
        N -= max(N-1, 0) // 14 * 14  # 14 is got from Solution2
        for i in xrange(N):
            cells = [0] + [cells[i-1] ^ cells[i+1] ^ 1 for i in xrange(1, 7)] + [0]
        return cells


# Time:  O(1)
# Space: O(1)
class Solution2(object):
    def prisonAfterNDays(self, cells, N):
        """
        :type cells: List[int]
        :type N: int
        :rtype: List[int]
        """
        cells = tuple(cells)
        lookup = {}
        while N:
            lookup[cells] = N
            N -= 1
            cells = tuple([0] + [cells[i - 1] ^ cells[i + 1] ^ 1 for i in xrange(1, 7)] + [0])
            if cells in lookup:
                assert(lookup[cells] - N in (1, 7, 14))
                N %= lookup[cells] - N
                break

        while N:
            N -= 1
            cells = tuple([0] + [cells[i - 1] ^ cells[i + 1] ^ 1 for i in xrange(1, 7)] + [0])
        return list(cells)
```

path-with-maximum-gold.py

```python
# In a gold mine grid of size m * n, each cell in this mine has an integer
# representing the amount of gold in that cell, 0 if it is empty.
#
# Return the maximum amount of gold you can collect under the conditions:
#
#
#       Every time you are located in a cell you will collect all the gold in
# that cell.
#       From your position you can walk one step to the left, right, up or down.
#       You can't visit the same cell more than once.
#       Never visit a cell with 0 gold.
#       You can start and stop collecting gold from any position in the grid
# that has some gold.
#
#
#
# Example 1:
#
# Input: grid = [[0,6,0],[5,8,7],[0,9,0]]
# Output: 24
# Explanation:
# [[0,6,0],
#  [5,8,7],
#  [0,9,0]]
# Path to get the maximum gold, 9 -> 8 -> 7.
#
#
# Example 2:
#
# Input: grid = [[1,0,7],[2,0,6],[3,4,5],[0,3,0],[9,0,20]]
# Output: 28
# Explanation:
# [[1,0,7],
#  [2,0,6],
#  [3,4,5],
#  [0,3,0],
#  [9,0,20]]
# Path to get the maximum gold, 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7.
#
#
#
# Constraints:
#
#
#       1 <= grid.length, grid[i].length <= 15
#       0 <= grid[i][j] <= 100
#       There are at most 25 cells containing gold.# Time:  O(m^2 * n^2)
# Space: O(m * n)

class Solution(object):
    def getMaximumGold(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        def backtracking(grid, i, j):
            result = 0
```

```
    grid[i][j] *= -1
    for dx, dy in directions:
        ni, nj = i+dx, j+dy
        if not (0 <= ni < len(grid) and
                0 <= nj < len(grid[0]) and
                grid[ni][nj] > 0):
            continue
        result = max(result, backtracking(grid, ni, nj))
    grid[i][j] *= -1
    return grid[i][j] + result

result = 0
for i in xrange(len(grid)):
    for j in xrange(len(grid[0])):
        if grid[i][j]:
            result = max(result, backtracking(grid, i, j))
return result
```

unique-binary-search-trees.py

```
# Given n, how many structurally unique BST's (binary search trees) that store
# values 1 ... n?
#
# Example:
#
# Input: 3
# Output: 5
# Explanation:
# 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
#
#
#
# Constraints:
#
#
#       1 <= n <= 19# Time:  O(n)
# Space: O(1)

class Solution(object):
    def numTrees(self, n):
        """
        :type n: int
        :rtype: int
        """
        if n == 0:
            return 1

        def combination(n, k):
            count = 1
            # C(n, k) = (n) / 1 * (n - 1) / 2 ... * (n - k + 1) / k
            for i in xrange(1, k + 1):
                count = count * (n - i + 1) / i
            return count

        return combination(2 * n, n) - combination(2 * n, n - 1)

# Time:  O(n^2)
# Space: O(n)
# DP solution.
class Solution2(object):
    # @return an integer
    def numTrees(self, n):
        counts = [1, 1]
        for i in xrange(2, n + 1):
            count = 0
            for j in xrange(i):
                count += counts[j] * counts[i - j - 1]
            counts.append(count)
        return counts[-1]
```

minimum-domino-rotations-for-equal-row.py

```python
# In a row of dominoes, A[i] and B[i] represent the top and bottom halves of the
# i-th domino.  (A domino is a tile with two numbers from 1 to 6 - one on each
# half of the tile.)
#
# We may rotate the i-th domino, so that A[i] and B[i] swap values.
#
# Return the minimum number of rotations so that all the values in A are the
# same, or all the values in B are the same.
#
# If it cannot be done, return -1.
#
#
#
# Example 1:
#
#
#
# Input: A = [2,1,2,4,2,2], B = [5,2,6,2,3,2]
# Output: 2
# Explanation:
# The first figure represents the dominoes as given by A and B: before we do any
# rotations.
# If we rotate the second and fourth dominoes, we can make every value in the
# top row equal to 2, as indicated by the second figure.
#
#
# Example 2:
#
# Input: A = [3,5,1,2,3], B = [3,6,3,3,4]
# Output: -1
# Explanation:
# In this case, it is not possible to rotate the dominoes to make one row of
# values equal.
#
#
#
#
# Note:
#
#
#       1 <= A[i], B[i] <= 6
#       2 <= A.length == B.length <= 20000# Time:  O(n)
# Space: O(1)

import itertools


class Solution(object):
    def minDominoRotations(self, A, B):
        """
        :type A: List[int]
        :type B: List[int]
        :rtype: int
        """
        intersect = reduce(set.__and__, [set(d) for d in itertools.izip(A, B)])
        if not intersect:
            return -1
        x = intersect.pop()
```

```
return min(len(A)-A.count(x), len(B)-B.count(x))
```

reveal-cards-in-increasing-order.py

```
# In a deck of cards, every card has a unique integer.  You can order the deck
# in any order you want.
#
# Initially, all the cards start face down (unrevealed) in one deck.
#
# Now, you do the following steps repeatedly, until all cards are revealed:
#
#
#       Take the top card of the deck, reveal it, and take it out of the deck.
#       If there are still cards in the deck, put the next top card of the deck
# at the bottom of the deck.
#       If there are still unrevealed cards, go back to step 1.  Otherwise,
# stop.
#
#
# Return an ordering of the deck that would reveal the cards in increasing
# order.
#
# The first entry in the answer is considered to be the top of the deck.
#
#
#
#
# Example 1:
#
# Input: [17,13,11,2,3,5,7]
# Output: [2,13,3,11,5,17,7]
# Explanation:
# We get the deck in the order [17,13,11,2,3,5,7] (this order doesn't matter),
# and reorder it.
# After reordering, the deck starts as [2,13,3,11,5,17,7], where 2 is the top of
# the deck.
# We reveal 2, and move 13 to the bottom.  The deck is now [3,11,5,17,7,13].
# We reveal 3, and move 11 to the bottom.  The deck is now [5,17,7,13,11].
# We reveal 5, and move 17 to the bottom.  The deck is now [7,13,11,17].
# We reveal 7, and move 13 to the bottom.  The deck is now [11,17,13].
# We reveal 11, and move 17 to the bottom.  The deck is now [13,17].
# We reveal 13, and move 17 to the bottom.  The deck is now [17].
# We reveal 17.
# Since all the cards revealed are in increasing order, the answer is correct.
#
#
#
#
#
# Note:
#
#
#       1 <= A.length <= 1000
#       1 <= A[i] <= 10^6
#       A[i] != A[j] for all i != j# Time:  O(n)
# Space: O(n)

import collections


class Solution(object):
    def deckRevealedIncreasing(self, deck):
```

```python
"""
:type deck: List[int]
:rtype: List[int]
"""
d = collections.deque()
deck.sort(reverse=True)
for i in deck:
    if d:
        d.appendleft(d.pop())
    d.appendleft(i)
return list(d)
```

fruit-into-baskets.py

```
# In a row of trees, the i-th tree produces fruit with type tree[i].
#
# You start at any tree of your choice, then repeatedly perform the following
# steps:
#
#
#       Add one piece of fruit from this tree to your baskets.  If you cannot,
# stop.
#       Move to the next tree to the right of the current tree.  If there is no
# tree to the right, stop.
#
#
# Note that you do not have any choice after the initial choice of starting
# tree: you must perform step 1, then step 2, then back to step 1, then step 2,
# and so on until you stop.
#
# You have two baskets, and each basket can carry any quantity of fruit, but you
# want each basket to only carry one type of fruit each.
#
# What is the total amount of fruit you can collect with this procedure?
#
#
#
# Example 1:
#
# Input: [1,2,1]
# Output: 3
# Explanation: We can collect [1,2,1].
#
#
#
# Example 2:
#
# Input: [0,1,2,2]
# Output: 3
# Explanation: We can collect [1,2,2].
# If we started at the first tree, we would only collect [0, 1].
#
#
#
# Example 3:
#
# Input: [1,2,3,2,2]
# Output: 4
# Explanation: We can collect [2,3,2,2].
# If we started at the first tree, we would only collect [1, 2].
#
#
#
# Example 4:
#
# Input: [3,3,3,1,2,1,1,2,3,3,4]
# Output: 5
# Explanation: We can collect [1,2,1,1,2].
# If we started at the first tree or the eighth tree, we would only collect 4
# fruits.
#
#
```

```
#
#
#
#
#
# Note:
#
#
#       1 <= tree.length <= 40000
#       0 <= tree[i] < tree.length# Time:  O(n)
# Space: O(1)

import collections


class Solution(object):
    def totalFruit(self, tree):
        """
        :type tree: List[int]
        :rtype: int
        """
        count = collections.defaultdict(int)
        result, i = 0, 0
        for j, v in enumerate(tree):
            count[v] += 1
            while len(count) > 2:
                count[tree[i]] -= 1
                if count[tree[i]] == 0:
                    del count[tree[i]]
                i += 1
            result = max(result, j-i+1)
        return result
```

binary-tree-preorder-traversal.py

```python
# Given a binary tree, return the preorder traversal of its nodes' values.
#
# Example:
#
# Input: [1,null,2,3]
#    1
#     \
#      2
#     /
#    3
#
# Output: [1,2,3]
#
#
# Follow up: Recursive solution is trivial, could you do it iteratively?# Time:  O(n)
# Space: O(1)

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


# Morris Traversal Solution
class Solution(object):
    def preorderTraversal(self, root):
        """
        :type root: TreeNode
        :rtype: List[int]
        """
        result, curr = [], root
        while curr:
            if curr.left is None:
                result.append(curr.val)
                curr = curr.right
            else:
                node = curr.left
                while node.right and node.right != curr:
                    node = node.right

                if node.right is None:
                    result.append(curr.val)
                    node.right = curr
                    curr = curr.left
                else:
                    node.right = None
                    curr = curr.right

        return result


# Time:  O(n)
# Space: O(h)
# Stack Solution
class Solution2(object):
    def preorderTraversal(self, root):
        """
```

```python
:type root: TreeNode
:rtype: List[int]
"""
result, stack = [], [(root, False)]
while stack:
    root, is_visited = stack.pop()
    if root is None:
        continue
    if is_visited:
        result.append(root.val)
    else:
        stack.append((root.right, False))
        stack.append((root.left, False))
        stack.append((root, True))
return result
```

reverse-words-in-a-string.py

```python
# Given an input string, reverse the string word by word.
#
#
#
# Example 1:
#
# Input: "the sky is blue"
# Output: "blue is sky the"
#
#
# Example 2:
#
# Input: "  hello world!  "
# Output: "world! hello"
# Explanation: Your reversed string should not contain leading or trailing
# spaces.
#
#
# Example 3:
#
# Input: "a good   example"
# Output: "example good a"
# Explanation: You need to reduce multiple spaces between two words to a single
# space in the reversed string.
#
#
#
#
# Note:
#
#
#       A word is defined as a sequence of non-space characters.
#       Input string may contain leading or trailing spaces. However, your
# reversed string should not contain leading or trailing spaces.
#       You need to reduce multiple spaces between two words to a single space
# in the reversed string.
#
#
#
#
# Follow up:
#
# For C programmers, try to solve it in-place in O(1) extra space.# Time:  O(n)
# Space: O(n)

class Solution(object):
    # @param s, a string
    # @return a string
    def reverseWords(self, s):
        return ' '.join(reversed(s.split()))
```

swap-nodes-in-pairs.py

```python
# Given a linked list, swap every two adjacent nodes and return its head.
#
# You may not modify the values in the list's nodes, only nodes itself may be
# changed.
#
#
#
# Example:
#
# Given 1->2->3->4, you should return the list as 2->1->4->3.# Time:  O(n)
# Space: O(1)

class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

    def __repr__(self):
        if self:
            return "{} -> {}".format(self.val, self.next)

class Solution(object):
    # @param a ListNode
    # @return a ListNode
    def swapPairs(self, head):
        dummy = ListNode(0)
        dummy.next = head
        current = dummy
        while current.next and current.next.next:
            next_one, next_two, next_three = current.next, current.next.next, current.next.next.next
            current.next = next_two
            next_two.next = next_one
            next_one.next = next_three
            current = next_one
        return dummy.next
```

convert-sorted-list-to-binary-search-tree.py

```python
# Given the head of a singly linked list where elements are sorted in ascending
# order, convert it to a height balanced BST.
#
# 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.
#
#
# Example 1:
#
# Input: head = [-10,-3,0,5,9]
# Output: [0,-3,9,-10,null,5]
# Explanation: One possible answer is [0,-3,9,-10,null,5], which represents the
# shown height balanced BST.
#
#
# Example 2:
#
# Input: head = []
# Output: []
#
#
# Example 3:
#
# Input: head = [0]
# Output: [0]
#
#
# Example 4:
#
# Input: head = [1,3]
# Output: [3,1]
#
#
#
# Constraints:
#
#
#       The numner of nodes in head is in the range [0, 2 * 10^4].
#       -10^5 <= Node.val <= 10^5# Time:  O(n)
# Space: O(logn)

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
#
# Definition for singly-linked list.
class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution(object):
    head = None
    # @param head, a list node
    # @return a tree node
    def sortedListToBST(self, head):
```

```python
        current, length = head, 0
        while current is not None:
            current, length = current.next, length + 1
        self.head = head
        return self.sortedListToBSTRecu(0, length)

    def sortedListToBSTRecu(self, start, end):
        if start == end:
            return None
        mid = start + (end - start) / 2
        left = self.sortedListToBSTRecu(start, mid)
        current = TreeNode(self.head.val)
        current.left = left
        self.head = self.head.next
        current.right = self.sortedListToBSTRecu(mid + 1, end)
        return current
```

shortest-path-in-binary-matrix.py

```
# Example 2:
#
# Input: [[0,0,0],[1,1,0],[1,1,0]]
#
#
# Output: 4# Time:  O(n^2)
# Space: O(n)

import collections


class Solution(object):
    def shortestPathBinaryMatrix(self, grid):
        """
        :type grid: List[List[int]]
        :rtype: int
        """
        directions = [(-1, -1), (-1, 0), (-1, 1), \
                      ( 0, -1), ( 0, 1), \
                      ( 1, -1), ( 1, 0), ( 1, 1)]
        result = 0
        q = collections.deque([(0, 0)])
        while q:
            result += 1
            next_depth = collections.deque()
            while q:
                i, j = q.popleft()
                if 0 <= i < len(grid) and \
                   0 <= j < len(grid[0]) and \
                    not grid[i][j]:
                    grid[i][j] = 1
                    if i == len(grid)-1 and j == len(grid)-1:
                        return result
                    for d in directions:
                        next_depth.append((i+d[0], j+d[1]))
            q = next_depth
        return -1
```

number-of-enclaves.py

```python
# Given a 2D array A, each cell is 0 (representing sea) or 1 (representing land)
#
# A move consists of walking from one land square 4-directionally to another
# land square, or off the boundary of the grid.
#
# Return the number of land squares in the grid for which we cannot walk off the
# boundary of the grid in any number of moves.
#
#
#
# Example 1:
#
# Input: [[0,0,0,0],[1,0,1,0],[0,1,1,0],[0,0,0,0]]
# Output: 3
# Explanation:
# There are three 1s that are enclosed by 0s, and one 1 that isn't enclosed
# because its on the boundary.
#
# Example 2:
#
# Input: [[0,1,1,0],[0,0,1,0],[0,0,1,0],[0,0,0,0]]
# Output: 0
# Explanation:
# All 1s are either on the boundary or can reach the boundary.
#
#
#
#
# Note:
#
#
#       1 <= A.length <= 500
#       1 <= A[i].length <= 500
#       0 <= A[i][j] <= 1
#       All rows have the same size.# Time:  O(m * n)
# Space: O(m * n)

class Solution(object):
    def numEnclaves(self, A):
        """
        :type A: List[List[int]]
        :rtype: int
        """
        directions = [(0, -1), (0, 1), (-1, 0), (1, 0)]
        def dfs(A, i, j):
            if not (0 <= i < len(A) and 0 <= j < len(A[0]) and A[i][j]):
                return
            A[i][j] = 0
            for d in directions:
                dfs(A, i+d[0], j+d[1])

        for i in xrange(len(A)):
            dfs(A, i, 0)
            dfs(A, i, len(A[0])-1)
        for j in xrange(1, len(A[0])-1):
            dfs(A, 0, j)
            dfs(A, len(A)-1, j)
        return sum(sum(row) for row in A)
```

word-search.py

```
# 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.
#
# Example:
#
# board =
# [
#   ['A','B','C','E'],
#   ['S','F','C','S'],
#   ['A','D','E','E']
# ]
#
# Given word = "ABCCED", return true.
# Given word = "SEE", return true.
# Given word = "ABCB", return false.
#
#
#
# Constraints:
#
#
#       board and word consists only of lowercase and uppercase English letters.
#       1 <= board.length <= 200
#       1 <= board[i].length <= 200
#       1 <= word.length <= 10^3# Time:  O(m * n * l)
# Space: O(l)

class Solution(object):
    # @param board, a list of lists of 1 length string
    # @param word, a string
    # @return a boolean
    def exist(self, board, word):
        visited = [[False for j in xrange(len(board[0]))] for i in xrange(len(board))]

        for i in xrange(len(board)):
            for j in xrange(len(board[0])):
                if self.existRecu(board, word, 0, i, j, visited):
                    return True

        return False

    def existRecu(self, board, word, cur, i, j, visited):
        if cur == len(word):
            return True

        if i < 0 or i >= len(board) or j < 0 or j >= len(board[0]) or visited[i][j] or board[i][j] != word[cur
            return False

        visited[i][j] = True
        result = self.existRecu(board, word, cur + 1, i + 1, j, visited) or\
                 self.existRecu(board, word, cur + 1, i - 1, j, visited) or\
                 self.existRecu(board, word, cur + 1, i, j + 1, visited) or\
                 self.existRecu(board, word, cur + 1, i, j - 1, visited)
        visited[i][j] = False
```

```
return result
```

iterator-for-combination.py

```python
# Design an Iterator class, which has:
#
#
#       A constructor that takes a string characters of sorted distinct
# lowercase English letters and a number combinationLength as arguments.
#       A function next() that returns the next combination of length
# combinationLength in lexicographical order.
#       A function hasNext() that returns True if and only if there exists a
# next combination.
#
#
#
#
# Example:
#
# CombinationIterator iterator = new CombinationIterator("abc", 2); // creates
# the iterator.
#
# iterator.next(); // returns "ab"
# iterator.hasNext(); // returns true
# iterator.next(); // returns "ac"
# iterator.hasNext(); // returns true
# iterator.next(); // returns "bc"
# iterator.hasNext(); // returns false
#
#
#
# Constraints:
#
#
#       1 <= combinationLength <= characters.length <= 15
#       There will be at most 10^4 function calls per test.
#       It's guaranteed that all calls of the function next are valid.# Time:  O(k), per operation
# Space: O(k)

import itertools


class CombinationIterator(object):

    def __init__(self, characters, combinationLength):
        """
        :type characters: str
        :type combinationLength: int
        """
        self.__it = itertools.combinations(characters, combinationLength)
        self.__curr = None
        self.__last = characters[-combinationLength:]

    def next(self):
        """
        :rtype: str
        """
        self.__curr = "".join(self.__it.next())
        return self.__curr

    def hasNext(self):
        """
```

```python
        :rtype: bool
        """
        return self.__curr != self.__last


# Time:  O(k), per operation
# Space: O(k)
import functools


class CombinationIterator2(object):

    def __init__(self, characters, combinationLength):
        """
        :type characters: str
        :type combinationLength: int
        """
        self.__characters = characters
        self.__combinationLength = combinationLength
        self.__it = self.__iterative_backtracking()
        self.__curr = None
        self.__last = characters[-combinationLength:]

    def __iterative_backtracking(self):
        def conquer():
            if len(curr) == self.__combinationLength:
                return curr

        def prev_divide(c):
            curr.append(c)

        def divide(i):
            if len(curr) != self.__combinationLength:
                for j in reversed(xrange(i, len(self.__characters)-(self.__combinationLength-len(curr)-1))):
                    stk.append(functools.partial(post_divide))
                    stk.append(functools.partial(divide, j+1))
                    stk.append(functools.partial(prev_divide, self.__characters[j]))
            stk.append(functools.partial(conquer))

        def post_divide():
            curr.pop()

        curr = []
        stk = [functools.partial(divide, 0)]
        while stk:
            result = stk.pop()()
            if result is not None:
                yield result

    def next(self):
        """
        :rtype: str
        """
        self.__curr = "".join(next(self.__it))
        return self.__curr

    def hasNext(self):
        """
        :rtype: bool
        """
```

```python
        return self.__curr != self.__last


# Your CombinationIterator object will be instantiated and called as such:
# obj = CombinationIterator(characters, combinationLength)
# param_1 = obj.next()
# param_2 = obj.hasNext()
```

complete-binary-tree-inserter.py

```python
# Example 1:
#
# Input: inputs = ["CBTInserter","insert","get_root"], inputs = [[[1]],[2],[]]
# Output: [null,1,[1,2]]
#
#
#
# Example 2:
#
# Input: inputs = ["CBTInserter","insert","insert","get_root"], inputs =
# [[[1,2,3,4,5,6]],[7],[8],[]]
# Output: [null,3,4,[1,2,3,4,5,6,7,8]]
#
#
#
#
#
# Note:
#
#
#       The initial given tree is complete and contains between 1 and 1000
# nodes.
#       CBTInserter.insert is called at most 10000 times per test case.
#       Every value of a given or inserted node is between 0 and 5000.# Time:  ctor:     O(n)
#        insert:   O(1)
#        get_root: O(1)
# Space: O(n)

class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class CBTInserter(object):

    def __init__(self, root):
        """
        :type root: TreeNode
        """
        self.__tree = [root]
        for i in self.__tree:
            if i.left:
                self.__tree.append(i.left)
            if i.right:
                self.__tree.append(i.right)

    def insert(self, v):
        """
        :type v: int
        :rtype: int
        """
        n = len(self.__tree)
        self.__tree.append(TreeNode(v))
        if n % 2:
            self.__tree[(n-1)//2].left = self.__tree[-1]
        else:
            self.__tree[(n-1)//2].right = self.__tree[-1]
```

```
        return self.__tree[(n-1)//2].val

    def get_root(self):
        """
        :rtype: TreeNode
        """
        return self.__tree[0]
```

print-foobar-alternately.py

```python
# Suppose you are given the following code:
#
# class FooBar {
#   public void foo() {
#     for (int i = 0; i < n; i++) {
#       print("foo");
#     }
#   }
#
#   public void bar() {
#     for (int i = 0; i < n; i++) {
#       print("bar");
#     }
#   }
# }
#
#
# The same instance of FooBar will be passed to two different threads. Thread A
# will call foo() while thread B will call bar(). Modify the given program to
# output "foobar" n times.
#
#
#
# Example 1:
#
# Input: n = 1
# Output: "foobar"
# Explanation: There are two threads being fired asynchronously. One of them
# calls foo(), while the other calls bar(). "foobar" is being output 1 time.
#
#
# Example 2:
#
# Input: n = 2
# Output: "foobarfoobar"
# Explanation: "foobar" is being output 2 times.# Time:  O(n)
# Space: O(1)

import threading


class FooBar(object):
    def __init__(self, n):
        self.__n = n
        self.__curr = False
        self.__cv = threading.Condition()

    def foo(self, printFoo):
        """
        :type printFoo: method
        :rtype: void
        """
        for i in xrange(self.__n):
            with self.__cv:
                while self.__curr != False:
                    self.__cv.wait()
                self.__curr = not self.__curr
                # printFoo() outputs "foo". Do not change or remove this line.
```

```
            printFoo()
            self.__cv.notify()

    def bar(self, printBar):
        """
        :type printBar: method
        :rtype: void
        """
        for i in xrange(self.__n):
            with self.__cv:
                while self.__curr != True:
                        self.__cv.wait()
                self.__curr = not self.__curr
                # printBar() outputs "bar". Do not change or remove this line.
                printBar()
                self.__cv.notify()
```

minimum-remove-to-make-valid-parentheses.py

```python
# Given a string s of '(' , ')' and lowercase English characters.
#
# Your task is to remove the minimum number of parentheses ( '(' or ')', in any
# positions ) so that the resulting parentheses string is valid and return any
# valid string.
#
# Formally, a parentheses string is valid if and only if:
#
#
#       It is the empty string, contains only lowercase characters, or
#       It can be written as AB (A concatenated with B), where A and B are valid
# strings, or
#       It can be written as (A), where A is a valid string.
#
#
#
# Example 1:
#
# Input: s = "lee(t(c)o)de)"
# Output: "lee(t(c)o)de"
# Explanation: "lee(t(co)de)" , "lee(t(c)ode)" would also be accepted.
#
#
# Example 2:
#
# Input: s = "a)b(c)d"
# Output: "ab(c)d"
#
#
# Example 3:
#
# Input: s = "))(("
# Output: ""
# Explanation: An empty string is also valid.
#
#
# Example 4:
#
# Input: s = "(a(b(c)d)"
# Output: "a(b(c)d)"
#
#
#
# Constraints:
#
#
#       1 <= s.length <= 10^5
#       s[i] is one of  '(' , ')' and lowercase English letters.# Time:  O(n)
# Space: O(n)

class Solution(object):
    def minRemoveToMakeValid(self, s):
        """
        :type s: str
        :rtype: str
        """
        result = list(s)
        count = 0
```

```python
for i, v in enumerate(result):
    if v == '(':
        count += 1
    elif v == ')':
        if count:
            count -= 1
        else:
            result[i] = ""
if count:
    for i in reversed(xrange(len(result))):
        if result[i] == '(':
            result[i] = ""
            count -= 1
            if not count:
                break
return "".join(result)
```

stone-game.py

```python
# Alex and Lee play a game with piles of stones.  There are an even number
# of piles arranged in a row, and each pile has a positive integer number of
# stones piles[i].
#
# The objective of the game is to end with the most stones.  The total number of
# stones is odd, so there are no ties.
#
# Alex and Lee take turns, with Alex starting first.  Each turn, a player takes
# the entire pile of stones from either the beginning or the end of the row.  This
# continues until there are no more piles left, at which point the person with the
# most stones wins.
#
# Assuming Alex and Lee play optimally, return True if and only if Alex wins the
# game.
#
#
# Example 1:
#
# Input: piles = [5,3,4,5]
# Output: true
# Explanation:
# Alex starts first, and can only take the first 5 or the last 5.
# Say he takes the first 5, so that the row becomes [3, 4, 5].
# If Lee takes 3, then the board is [4, 5], and Alex takes 5 to win with 10
# points.
# If Lee takes the last 5, then the board is [3, 4], and Alex takes 4 to win
# with 9 points.
# This demonstrated that taking the first 5 was a winning move for Alex, so we
# return true.
#
#
#
# Constraints:
#
#
#       2 <= piles.length <= 500
#       piles.length is even.
#       1 <= piles[i] <= 500
#       sum(piles) is odd.# Time:  O(n^2)
# Space: O(n)

class Solution(object):
    def stoneGame(self, piles):
        """
        :type piles: List[int]
        :rtype: bool
        """
        if len(piles) % 2 == 0 or len(piles) == 1:
            return True

        dp = [0] * len(piles)
        for i in reversed(xrange(len(piles))):
            dp[i] = piles[i]
            for j in xrange(i+1, len(piles)):
                dp[j] = max(piles[i] - dp[j], piles[j] - dp[j - 1])
        return dp[-1] >= 0
```

longest-substring-without-repeating-characters.py

```
# Example 1:
#
# Input: "abcabcbb"
# Output: 3
# Explanation: The answer is "abc", with the length of 3.
#
#
#
# Example 2:
#
# Input: "bbbbb"
# Output: 1
# Explanation: The answer is "b", with the length of 1.
#
#
#
# Example 3:
#
# Input: "pwwkew"
# Output: 3
# Explanation: 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.# Time:  O(n)
# Space: O(1)

class Solution(object):
    def lengthOfLongestSubstring(self, s):
        """
        :type s: str
        :rtype: int
        """
        result, left = 0, 0
        lookup = {}
        for right in xrange(len(s)):
            if s[right] in lookup:
                left = max(left, lookup[s[right]]+1)
            lookup[s[right]] = right
            result = max(result, right-left+1)
        return result
```

combination-sum-ii.py

```
# Given a collection of candidate numbers (candidates) and a target number
# (target), find all unique combinations in candidates where the candidate numbers
# sums to target.
#
# Each number in candidates 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.
#
#
# Example 1:
#
# Input: candidates = [10,1,2,7,6,1,5], target = 8,
# A solution set is:
# [
#   [1, 7],
#   [1, 2, 5],
#   [2, 6],
#   [1, 1, 6]
# ]
#
#
# Example 2:
#
# Input: candidates = [2,5,2,1,2], target = 5,
# A solution set is:
# [
#   [1,2,2],
#   [5]
# ]# Time:  O(k * C(n, k))
# Space: O(k)

class Solution(object):
    # @param candidates, a list of integers
    # @param target, integer
    # @return a list of lists of integers
    def combinationSum2(self, candidates, target):
        result = []
        self.combinationSumRecu(sorted(candidates), result, 0, [], target)
        return result

    def combinationSumRecu(self, candidates, result, start, intermediate, target):
        if target == 0:
            result.append(list(intermediate))
        prev = 0
        while start < len(candidates) and candidates[start] <= target:
            if prev != candidates[start]:
                intermediate.append(candidates[start])
                self.combinationSumRecu(candidates, result, start + 1, intermediate, target - candidates[start
                intermediate.pop()
                prev = candidates[start]
            start += 1
```

task-scheduler.py

```python
# Given a characters array tasks, representing the tasks a CPU needs to do,
# where each letter represents a different task. Tasks could be done in any order.
# Each task is done in one unit of time. For each unit of time, the CPU could
# complete either one task or just be idle.
#
# However, there is a non-negative integer n that represents the cooldown period
# between two same tasks (the same letter in the array), that is that there must
# be at least n units of time between any two same tasks.
#
# Return the least number of units of times that the CPU will take to finish all
# the given tasks.
#
#
# Example 1:
#
# Input: tasks = ["A","A","A","B","B","B"], n = 2
# Output: 8
# Explanation:
# A -> B -> idle -> A -> B -> idle -> A -> B
# There is at least 2 units of time between any two same tasks.
#
#
# Example 2:
#
# Input: tasks = ["A","A","A","B","B","B"], n = 0
# Output: 6
# Explanation: On this case any permutation of size 6 would work since n = 0.
# ["A","A","A","B","B","B"]
# ["A","B","A","B","A","B"]
# ["B","B","B","A","A","A"]
# ...
# And so on.
#
#
# Example 3:
#
# Input: tasks = ["A","A","A","A","A","A","B","C","D","E","F","G"], n = 2
# Output: 16
# Explanation:
# One possible solution is
# A -> B -> C -> A -> D -> E -> A -> F -> G -> A -> idle -> idle -> A -> idle ->
# idle -> A
#
#
#
# Constraints:
#
#
#       1 <= task.length <= 104
#       tasks[i] is upper-case English letter.
#       The integer n is in the range [0, 100].# Time:  O(n)
# Space: O(26) = O(1)

from collections import Counter


class Solution(object):
    def leastInterval(self, tasks, n):
```

```python
"""
:type tasks: List[str]
:type n: int
:rtype: int
"""
counter = Counter(tasks)
_, max_count = counter.most_common(1)[0]

result = (max_count-1) * (n+1)
for count in counter.values():
    if count == max_count:
        result += 1
return max(result, len(tasks))
```

find-minimum-in-rotated-sorted-array.py

```python
# Suppose an array sorted in ascending order 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.
#
# Example 1:
#
# Input: [3,4,5,1,2]
# Output: 1
#
#
# Example 2:
#
# Input: [4,5,6,7,0,1,2]
# Output: 0# Time:  O(logn)
# Space: O(1)

class Solution(object):
    def findMin(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        left, right = 0, len(nums)
        target = nums[-1]

        while left < right:
            mid = left + (right - left) / 2

            if nums[mid] <= target:
                right = mid
            else:
                left = mid + 1

        return nums[left]


class Solution2(object):
    def findMin(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        left, right = 0, len(nums) - 1
        while left < right and nums[left] >= nums[right]:
            mid = left + (right - left) / 2

            if nums[mid] < nums[left]:
                right = mid
            else:
                left = mid + 1

        return nums[left]
```

maximum-width-of-binary-tree.py

```
# Given a binary tree, write a function to get the maximum width of the given
# tree. The maximum width of a tree is the maximum width among all levels.
#
# The width of one level is defined as the length between the end-nodes (the
# leftmost and right most non-null nodes in the level, where the null nodes
# between the end-nodes are also counted into the length calculation.
#
# It is guaranteed that the answer will in the range of 32-bit signed integer.
#
# Example 1:
#
# Input:
#
#            1
#          /   \
#         3     2
#        / \     \
#       5   3     9
#
# Output: 4
# Explanation: The maximum width existing in the third level with the length 4
# (5,3,null,9).
#
#
# Example 2:
#
# Input:
#
#           1
#          /
#         3
#        / \
#       5   3
#
# Output: 2
# Explanation: The maximum width existing in the third level with the length 2
# (5,3).
#
#
# Example 3:
#
# Input:
#
#           1
#          / \
#         3   2
#        /
#       5
#
# Output: 2
# Explanation: The maximum width existing in the second level with the length 2
# (3,2).
#
#
# Example 4:
#
# Input:
#
```

```python
#           1
#          / \
#         3   2
#        /     \
#       5       9
#      /         \
#     6           7
# Output: 8
# Explanation:The maximum width existing in the fourth level with the length 8
# (6,null,null,null,null,null,null,7).
#
#
#
# Constraints:
#
#
#       The given binary tree will have between 1 and 3000 nodes.# Time:  O(n)
# Space: O(h)

class Solution(object):
    def widthOfBinaryTree(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        def dfs(node, i, depth, leftmosts):
            if not node:
                return 0
            if depth >= len(leftmosts):
                leftmosts.append(i)
            return max(i-leftmosts[depth]+1, \
                       dfs(node.left, i*2, depth+1, leftmosts), \
                       dfs(node.right, i*2+1, depth+1, leftmosts))

        leftmosts = []
        return dfs(root, 1, 0, leftmosts)
```

fraction-addition-and-subtraction.py

```python
# Given a string representing an expression of fraction addition and
# subtraction, you need to return the calculation result in string format. The
# final result should be irreducible fraction. If your final result is an integer,
# say 2, you need to change it to the format of fraction that has denominator 1.
# So in this case, 2 should be converted to 2/1.
#
# Example 1:
#
# Input:"-1/2+1/2"
# Output: "0/1"
#
#
#
# Example 2:
#
# Input:"-1/2+1/2+1/3"
# Output: "1/3"
#
#
#
# Example 3:
#
# Input:"1/3-1/2"
# Output: "-1/6"
#
#
#
# Example 4:
#
# Input:"5/3+1/3"
# Output: "2/1"
#
#
#
# Note:
#
#
# The input string only contains '0' to '9', '/', '+' and '-'. So does the
# output.
# Each fraction (input and output) has format ±numerator/denominator. If the
# first input fraction or the output is positive, then '+' will be omitted.
# The input only contains valid irreducible fractions, where the numerator and
# denominator of each fraction will always be in the range [1,10]. If the
# denominator is 1, it means this fraction is actually an integer in a fraction
# format defined above.
# The number of given fractions will be in the range [1,10].
# The numerator and denominator of the final result are guaranteed to be valid
# and in the range of 32-bit int.# Time:  O(nlogx), x is the max denominator
# Space: O(n)

import re


class Solution(object):
    def fractionAddition(self, expression):
        """
        :type expression: str
        :rtype: str
```

```
"""
def gcd(a, b):
    while b:
        a, b = b, a%b
    return a

ints = map(int, re.findall('[+-]?\d+', expression))
A, B = 0, 1
for i in xrange(0, len(ints), 2):
    a, b = ints[i], ints[i+1]
    A = A * b + a * B
    B *= b
    g = gcd(A, B)
    A //= g
    B //= g
return '%d/%d' % (A, B)
```